

图灵程序设计丛书



ng-book 2

The Complete Book on Angular 2

Angular 权威教程

[美] Ari Lerner [巴西] Felipe Coury
[美] Nate Murray [巴西] Carlos Taborda 著
Nice Angular社区 译

强力的IDE支持+完善的生态圈+一套代码、多种平台+……

= Angular

资深全栈开发工程师经验总结
雪狼组织的Nice Angular社区主力倾情翻译

Google推荐阅读

数字版权声明

Ari Lerner

全栈工程师，拥有多年Angular经验，自办并运营Angular电子报ng-newsletter.com，在著名硅谷工程师培训学校Hack Reactor担任AngularJS讲师。Fullstack.io创始人。

Felipe Coury

Gistia Labs联合创始人兼CTO。

Nate Murray

全栈工程师，曾任职于IFTTT，拥有数据挖掘和增量Web服务等方面的背景。

Carlos Taborda

Gistia Labs联合创始人兼主管。

TURING 图灵程序设计丛书



ng-book 2

The Complete Book on Angular 2

Angular权威教程

[美] Ari Lerner [巴西] Felipe Coury
[美] Nate Murray [巴西] Carlos Taborda 著
Nice Angular社区 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Angular权威教程 / （美）阿里·勒纳（Ari Lerner）等著；Nice Angular社区译. -- 2版. -- 北京：人民邮电出版社，2017.4
（图灵程序设计丛书）
ISBN 978-7-115-45158-3

Ⅰ. ①A… Ⅱ. ①阿… ②N… Ⅲ. ①超文本标记语言－程序设计－教材 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第051224号

内 容 提 要

本书堪称 Angular 领域的里程碑式著作，涵盖了关于 Angular 的几乎所有内容。对于没有经验的人，本书平实、通俗的讲解，递进、严密的组织，可以让人毫无压力地登堂入室，迅速领悟新一代 Web 应用开发的精髓。如果你有相关经验，那本书对 Angular 概念和技术细节的全面剖析，以及引人入胜、切中肯綮的讲解，将帮助你彻底掌握这个框架，在自己职业技术修炼之路上更进一步。

本书的读者对象为所有想要理解和学习 Angular 的前端开发人员。

◆ 著　　[美] Ari Lerner　[巴西] Felipe Coury
　　　　[美] Nate Murray　[巴西] Carlos Taborda
译　　Nice Angular社区
责任编辑　杨　琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京印刷

◆ 开本：800×1000　1/16
印张：32
字数：756千字　　2017年4月第2版
印数：14 301 – 18 300册　　2017年4月北京第1次印刷

著作权合同登记号　图字：01-2017-0675号

定价：109.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版 权 声 明

推 荐 序

很高兴这本《Angular权威教程》成为Angular中文资源的一部分，希望它能广受欢迎，给中国的Angular社区提供一份令人愉悦的学习资源，也希望它帮助更多工程师开始使用下一代Angular框架来开发应用。

我认识雪狼和他所属的Nice Angular社区是在2016年。那时候，他们开始了对Angular官方网站卓越的本地化工作。现在，这份中文官网已经部署在了angular.cn上。

本书及其翻译工作充分体现了中国开源软件开发者的热情和共享精神。感谢雪狼等来自Nice Angular社区的志愿者们对此作出的贡献。愿本书帮助你开始试用Angular！祝你成功！

Naomi Black，Google Angular项目经理兼主管

作为一项开源技术和前沿Web开发框架，Angular持续吸引着中国区开发人员的关注。作为雪狼及其所属Nice Angular社区的集体工作成果，这本书是开源力量的又一次证明，证明这种热情、这种志愿精神确实可以帮助业界享受到全球最新的开发技术。我谨代表Google开发技术推广部向这本书的出版表示祝贺。

栾跃，Google开发技术推广部大中华区主管

译　者　序

简介

以笔者之所见，《Angular权威教程》大概是目前除了Angular官方文档之外最全面的学习资料了，这从其英文版多达600多页的篇幅就可见一斑。相应地，它面对的对象涵盖了从入门级到中高级的读者，是一本可以陪伴你成长的好书。

在内容安排上，本书具有大量的例子以保障其足够浅显，但也穿插着一些原理分析以保障其足够深入。除此之外，本书还给出了很多外部参考资料，让富有探险精神的你可以向专家级进发。

翻译说明

未来的版本号及发布计划

Angular就要出4.0了！是的，过一阵子还有Angular 5/6/7/8……这本书会很快过时吗？答案是“不会”。Angular开发组对于未来的版本号及发布计划有一个正式的说明，大意是：

> 我们要兼顾向后兼容和向前演进，因此以后我们将严格遵循SemVer语义化版本规范，并力求让版本升级变得可预测，以便使用者可以提前安排。在大版本号之间会出现少量破坏性变更，但是不用担心，相邻的大版本号之间只会把一些API标记为废弃的。也就是说，理想情况下，4的程序是可以直接迁移到5的，只是会收到一些API废弃提示，到6中才会彻底移除。同时，官方会在文档中给出详细的升级指南，帮助开发者升级。

因此，尽请放心，Angular以后绝不会出现像从1升级到2这么大的变化。事实上，NodeJS现在采用的就是类似的版本策略，提高发布的可预测性对于工程化开发是很有价值的。

另外，这里为什么没有3？简单点说就是因为路由模块比其他模块多发布过一次，因此当你使用core模块的2.0时，和它配套的router模块却是3.0的，这容易让开发人员困惑，跳过3，可以让所有模块的编号重新对齐。

对框架名称的说明

Angular开发组正式确定了新的命名策略：用AngularJS来代表1.x版本，而Angular代表2.x、4.x、5.x等很多后续版本，因为Angular 2+将支持TypeScript/JavaScript/Dart，而不再是JavaScript。这些变化已经在官方文档中体现出来了，而本书也将同样遵循这样的命名策略。

名词：装饰器与注解

`@Component`等语法元素在TypeScript中被称为装饰器（decorator），但在本书中，作者统一称其为注解（annotation）。这两种提法都是正确的。在语法层面，`@Component`确实是装饰器，这是TypeScript的标准叫法；但是在语义层面，Angular中是把它作为注解使用的。两者的区别是，装饰器直接改变被装饰者的行为，而注解则提供元数据，供框架去根据这些元数据做不同的处理。在Angular目前的版本中，`@Component`确实只是提供了元数据。

我们在跟原作者讨论之后，决定还是跟随作者的提法来翻译。不过在日常工作中，还是建议你遵循TypeScript的提法，将其称为装饰器。

支持与勘误

如果对本书中的一些概念不太理解，请参阅Angular官方中文站angular.cn。这里有来自官方开发组的权威资料。

如果对本书有任何疑问或发现问题，请到https://github.com/nice-angular/ng-book-2提交issue。

同时，对于一些经过确认的issue，我们也会更新在勘误区。

关于我们

参与本次翻译的一共有7位成员，都是AngularJS领域的专家和Angular领域的先行者。稍后会有我们的简短介绍。

本书各章的译者和校对者如下：

	翻译	一校	二校
第1章	雪狼、叶志敏	郑丰彧	郑丰彧
第2章	破狼	破狼	雪狼
第3章	张旋	张旋	雪狼
第4章	郑丰彧	郑丰彧	雪狼
第5章	破狼	破狼	雪狼
第6章	王子实	王子实	雪狼

（续）

	翻译	一校	二校
第7章	叶志敏	叶志敏	叶志敏
第8章	雪狼	雪狼	雪狼
第9章	郑丰彧	郑丰彧	雪狼
第10章	郑丰彧	郑丰彧	Hantsy
第11章	郑丰彧	郑丰彧	Hantsy
第12章	郑丰彧	郑丰彧	雪狼
第13章	郑丰彧	郑丰彧	雪狼
第14章	郑丰彧	郑丰彧	雪狼
第15章	Hantsy	Hantsy	叶志敏
第16章	雪狼	雪狼	张旋

除此之外，雪狼还承担了项目管理和中文统稿工作；破狼负责全书的技术准确性把关；叶志敏负责与作者沟通，并在英文理解方面进行把关。

我们的感恩

本书得以发行，首先要感谢Angular开发组及其项目经理Naomi Black。正是由于她的支持和牵线搭桥，才有了我们和图灵的这次合作。

我们还要感谢Google开发技术推广部及其大中华区主管栾跃和项目经理程路，正是由于他们的努力，让Angular在中国的推广普及工作有了正规军的加入，而本书的出版正是推广计划中的一小部分。

我们还要感谢图灵的编辑朱巍和杨琳，在整个翻译过程中，她们给了我们许多专业的指导和帮助。本书得以在迅速出版的同时保证高质量，她们的经验和把关居功甚伟。

最后，要感谢Angular中文社区。我所指的并不是由我们几个创建并管理的这些QQ群、微信群等，而是指广义的中文社区。无论你在北京还是上海，也无论你在国内还是海外；无论你是高手还是新兵，也无论你是否像我们一样是Angular的忠实粉丝，你们都是广义Angular中文社区中的一员。在我们的心中，只有一个Angular中文社区，她不被任何人拥有，也被每一个人拥有，因为她就是我们每个人。

固然，我们这几位译者都是推广Angular的志愿者与先行者，但我们真正希望看到的是一个繁荣、开放、互通的中文社区，是全球Angular社区的一部分，我们希望看到Angular的技术社区遍地开花。因此，如果你有自己的组织或影响力，请联系我们，我们愿与你携手共进，分享各种知识、渠道与资源，共同制定与推进社区发展计划。要知道，无论你将来是求职还是创业，一个繁荣的社区都会给你带来强力的支持。

一旦有了共同的愿景和开放、包容的文化，我们就能无视时空的阻隔，在天南海北守望相助，共同面对新技术的挑战与机遇。纷繁的世界、冰冷的技术与温暖的社区，共同构成了本书的出版背景。

雪狼的感恩

汪志成，网名雪狼。ThoughtWorker & Google开发者专家（GDE），拥有18年软件开发经验，崇尚简单、专业、分享，“好为人师，好为人师”；合著有《AngularJS深度剖析与最佳实践》。

首先，我要感谢我的家人，特别是我的妻子春娜。为了翻译官方文档和这本书，我失去了很多陪伴他们的时间，没有他们的支持，故事将无从开始。

其次，我要感谢ThoughtWorks，没有这样一个平台，我就无法安心钻研技术，更没有大量把新技术应用于工程实践中的机会。

最后，要特别感谢我刚刚出生的女儿，你是激励我前进的动力。闺女，看到了吗？这是老爹给你的迎新礼物。

破狼的感恩

格茸扎西，网名破狼。ThoughtWorks一线码农、架构师、咨询师；爱好读书和旅游，也常涂鸦一些技术博文；合著有《AngularJS深度剖析与最佳实践》；国内Angular最早布道者，Nice Angular社区“狼主”。

首先，要感谢我的妻子和父亲。因为他们的鼓励，我才能顺利完成本书相应章节的翻译。

其次，要感谢ThoughtWorks这个大家庭。因为在这个自组织和黑客文化环境的熏陶下，我才能潜心钻研这些技术。

最后，要感谢图灵出版社的朱巍编辑、本书的作者以及其他译者们。

叶志敏的感恩

叶志敏，虽留英多年、远漂他乡、四处奔波，一颗热爱软件开发的心却依旧如初。多年前曾与雪狼共事，合作愉快，因此成为好朋友。由雪狼推荐进入Angular世界，使用Angular和.NET平台开发软件多年。从Alpha阶段开始使用Angular。与雪狼合作，翻译Angular官方文档站，并经过Angular团队的推荐，承接翻译本书的重任。

首先感谢我的妻子。从怀孕到照顾女儿健康成长，她一直对我的工作非常理解和支持，从无怨言。其次，感谢我母亲和岳母的慈爱与帮助。最后，希望女儿能健康成长，平安一生。

Hantsy 的感恩

Hantsy，拥有15年软件工程经验。2012年曾受JBoss（RedHat子公司）邀请前往波士顿参加JBoss用户和开发人员年度大会，并获得JBoss Community Recognition Awards。现为自由职业者，远程工作多年。

感谢Angular中文团队和图灵的支持，非常荣幸参与本书中文版的翻译。感谢Angular团队的努力，为我们带来如此优秀的工具框架。

张旋的感恩

张旋，PMP、ACP、NPDP，中科院计算所烟台分所集成应用中心主任。1982年生人，1996年起接触编程。正式从事软件工作行业11年。擅长项目管理、团队管理、技术体系建设。非常喜欢研究和对比各种新技术，生成适合工程使用的技术栈，并灌输到整个团队中去。

十分荣幸能成为Nice Angular社区的一员，感谢雪狼和破狼。感谢本书原作者为我们提供了一本这么好的Angular教程，也感谢本书的所有翻译者，从你们身上我确实学到了很多。感谢我的老婆莉莉，照看乐乐辛苦了，谢谢你给我时间让我做自己喜欢的事。最后感谢图灵出版社的朱巍编辑，本次合作非常愉快，期待下次更好的机会！

郑丰彧的感恩

郑丰彧，网名Z，现就职于大商集团天狗网，Angular爱好者，喜欢函数式编程、WebGL。

首先，我要感谢雪狼，一次很偶然的机会受到雪狼的邀请，让我受宠若惊，也为我开启了这次Angular翻译之旅。

其次，我要感谢我的家人，尤其是有孕在身的老婆和孕育中的宝宝。为了翻译这本书，我牺牲了很多原本用来陪伴你们的时间。

最后，我想对即将出世的女儿柚柚说句话：我们全家人对你的期待正如我们Nice Angular社区对此书的期待。所以，赶快“问世”吧！

王子实的感恩

王子实，现任光辉城市全栈工程师。1992年生，自学生时代便喜好编程，一直以来对各种新技术非常着迷，乐于对其进行研究与探索，并将成果在团队中进行推广，以提

升整体效率。

非常感谢雪狼能够给我这次机会参与到本书的翻译中来，能够让我对Angular社区尽一点点自己的绵薄之力。

同时也要感谢其他参与翻译的译者们，让我有了这次非常宝贵的经验。尤其是在翻译过程中遇到一些技术问题以及对原书内容有一些疑惑时，大家探究与实践的精神让我印象深刻。

还要感谢我的妻子默默给予我支持与理解。

最后，就是要感谢Google带给我们Angular这个强大而又好用的框架。希望它也能越来越好，不断进步！

目　　录

第 1 章

编写你的第一个Angular Web应用

1

1.1　仿制 Reddit 网站

在本章中，我们将构建一个应用，它能让用户发表推荐文章（包括标题和URL）并对每篇文章投票。

你可以把该应用看作类似于Reddit[1]或Product Hunt[2]的起步版网站。

这个简单的应用将涵盖Angular中的大部分基本要素，包括：

- 构建自定义组件；
- 从表单中接收用户输入；
- 把对象列表渲染到视图中；
- 拦截用户的点击操作，并据此作出反应。

读完本章之后，你将掌握如何构建基本的Angular应用。

图1-1展示了该应用最终完成后的界面截图。

首先，用户将提交一个新的链接。之后，其他用户可以对每篇文章投票："顶"或"踩"。每个链接都有一个最终得票数，我们可以对自己认为有用的链接投票（如图1-2所示）。

① http://reddit.com

② http://producthunt.com

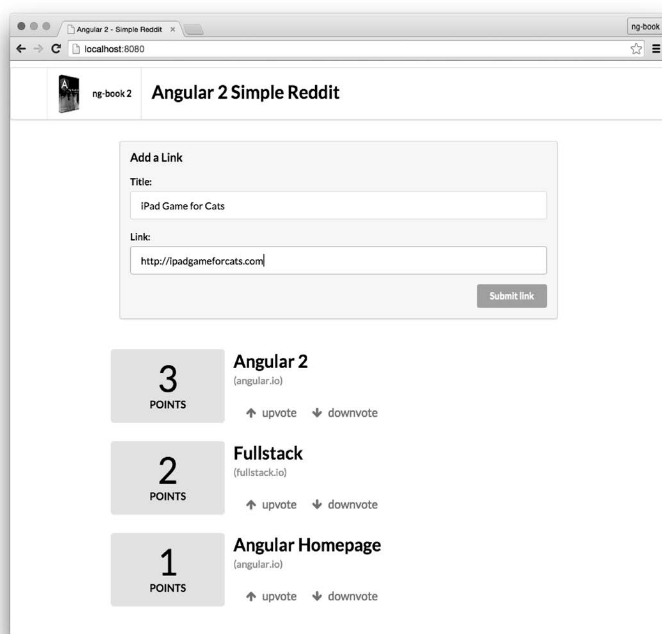


图1-1　完成后的应用

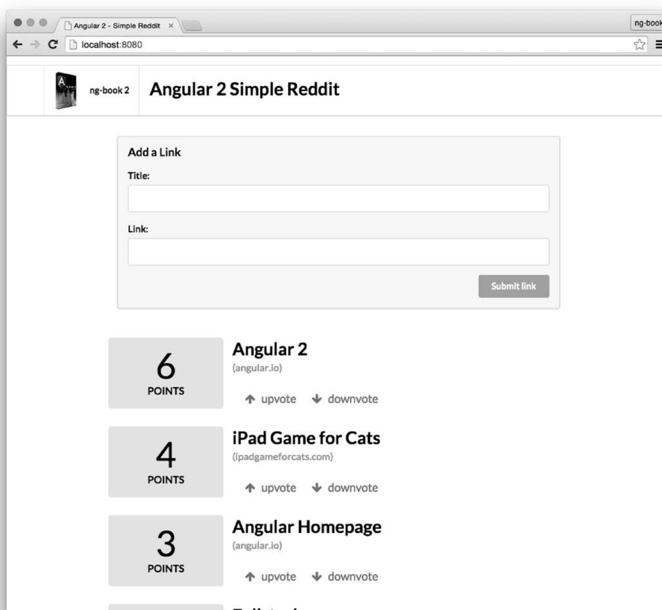


图1-2　包含新文章的应用

在本项目和整本书中，我们都将使用TypeScript。TypeScript是JavaScript ES6版的一个超集，增加了类型支持。本章不会深入讲解TypeScript；如果你熟悉ES5（“普通”的JavaScript）或ES6（ES2015），那么在后续的学习过程中应该不会有什么问题。

在第2章中，我们将更深入地学习TypeScript。因此，即使你对某些新语法不太熟悉，也不必担心。

1.2　起步

1.2.1　TypeScript

要开始使用TypeScript，首先需要安装Node.js。安装方式很多，请参见Node.js官方网站（https://nodejs.org/download/）了解详情。



我必须用TypeScript吗？并非如此！要使用Angular，TypeScript并不是必需的，但它可能是最好的选择。Angular也有一套ES5 API，但Angular本身就是用TypeScript写成的，所以人们一般也会选用它。本书也将使用TypeScript，因为它确实很棒，能让Angular写起来更简单。当然，并不是非它不可。

安装完Node.js，接着就要安装TypeScript了。请确保安装1.7或更高的版本。要想安装它，请运行下列npm命令：

```
$ npm install -g typescript
```



通常，npm是Node.js的一部分。如果你的系统中没有npm命令，请确认你安装的Node.js是包含它的版本。



Windows用户：我们将在全书中使用Linux/Mac风格的命令行。强烈建议你安装Cygwin[①]。借助它，你就能直接运行本书中的这些命令了。

1.2.2　`angular-cli`

Angular提供了一个命令行工具`angular-cli`，它能让用户通过命令行创建和管理项目。它自动化了一系列任务，比如创建项目、添加新的控制器等。多数情况下，选用`angular-cli`都是明

① https://www.cygwin.com/

智的决定。当你创建和维护应用时，它能帮你遵循很多常用模式。

要想安装angular-cli，只要运行下列命令即可：

```
$ npm install -g angular-cli@1.0.0-beta.18
```

安装完毕之后，你就可以在命令行中用ng命令运行它了。运行ng命令时，你会看到一大堆输出，不过不用管它；往回滚屏，你会看到如下内容：

```
$ ng
Could not start watchman; falling back to NodeWatcher for file system events.
Visit http://ember-cli.com/user-guide/#watchman for more info.
Usage: ng <command (Default: help)>
```

之所以得到这一大堆输出，是因为当我们不带参数运行ng命令时，它就会执行默认的help命令。help命令会解释如何使用本工具。

如果你在OS X或Linux上运行，可能还会在输出中看到这一行：

```
Could not start watchman; falling back to NodeWatcher for file system events.
```

这意味着我们没有安装过一个名叫watchman的工具。此工具能帮助angular-cli监听文件系统的变化。如果你在OS X上运行，建议使用Homebrew工具安装它，命令如下：

```
$ brew install watchman
```



如果你是OS X用户并且运行这个brew命令时出现错误，那么表示你尚未正确安装Homebrew工具。请参阅http://brew.sh/来安装它，然后再试一次。

如果你是Linux用户，可以参阅https://ember-cli.com/user-guide/#watchman来学习如何安装watchman。

如果你是Windows用户，那么不必安装任何东西，angular-cli将使用原生的Node.js文件监视器。

现在你应该已经装好angular-cli及其依赖了。在本章中，我们就用它来创建第一个应用。

1.2.3　示例项目

现在，环境已经准备好了，我们这就来编写第一个Angular应用吧！

打开终端窗口并且运行ng new命令，快速创建一个新的项目：

```
$ ng new angular2_hello_world
```

运行之后，你将看到下列输出：

```
installing ng 2
  create .editorconfig
  create README.md
```

```
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/app/index.ts
  create src/app/shared/index.ts
  create src/assets/.gitkeep
  create src/assets/.npmignore
  create src/environments/environment.dev.ts
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create src/typings.d.ts
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
 Installing packages for tooling via npm
```

它将运行一段时间，进行npm依赖的安装。一旦安装结束，我们会看到一条成功信息：

```
Installed packages for tooling via npm.
```

这里生成了很多文件！现在不用关心它们都是什么。我们会在本书中讲解每一个文件的含义和用途。不过现在，我们先把注意力集中在如何用Angular代码开始工作上。

进入ng命令创建的angular2_hello_world目录，来看看它里面都有什么：

```
$ cd angular2_hello_world
$ tree -F -L 1
.
├── README.md            // an useful README
├── angular-cli.json     // angular-cli configuration file
├── e2e/                 // end to end tests
├── karma.conf.js        // unit test configuration
├── node_modules/        // installed dependencies
├── package.json         // npm configuration
├── protractor.conf.js   // e2e test configuration
├── src/                 // application source
└── tslint.json          // linter config file
```

```
3 directories, 6 files
```

我们目前关注的目录是src，应用代码就在里面。下面看看我们在那里创建了什么：

```
$ cd src
$ tree -F
.
|-- app/
|   |-- app.component.css
|   |-- app.component.html
|   |-- app.component.spec.ts
|   |-- app.component.ts
|   |-- app.module.ts
|   |-- index.ts
|   `-- shared/
|       `-- index.ts
|-- assets/
|-- environments/
|   |-- environment.dev.ts
|   |-- environment.prod.ts
|   `-- environment.ts
|-- favicon.ico
|-- index.html
|-- main.ts
|-- polyfills.ts
|-- styles.css
|-- test.ts
|-- tsconfig.json
`-- typings.d.ts

4 directories, 18 files
```

用你惯用的文本编辑器打开index.html，应该会看到如下代码。

code/first_app/angular2_hello_world/src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular2HelloWorld</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

我们把它分解一下。

code/first_app/angular2_hello_world/src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular2HelloWorld</title>
  <base href="/">
```

如果你熟悉HTML，这第一部分就很平淡无奇了。我们在这里声明了页面的字符集（charset）、标题（title）和基础URL（base href）。

code/first_app/angular2_hello_world/src/index.html

```
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

如果你继续深入模板主体（body），就会看到下列代码。

code/first_app/angular2_hello_world/src/index.html

```
  <app-root>Loading...</app-root>
</body>
</html>
```

我们的应用将会在app-root标签处进行渲染，稍后剖析源代码的其他部分时还会看到它。文本Loading...是一个占位符，在应用代码加载之前会显示它。我们可以借助此技巧来通知用户该应用正在加载，可以像这里一样显示一条消息，也可以显示一个加载动画或其他形式的进度通知。

之后就可以编写应用代码了。

1.3 运行应用

在开始修改之前，我们先把这个自动生成的初始应用加载到浏览器中。angular-cli有一个内建的HTTP服务器，我们可以用它来启动应用。回到终端，进入应用的根目录（在本应用中是./angular2_hello_world目录）并运行命令。

```
$ ng serve
** NG Live Development Server is running on http://localhost:4200. **
// a bunch of debug messages

Build successful - 1342ms.
```

我们的应用正在localhost的4200端口上运行。打开浏览器并访问http://localhost:4200，结果如图1-3所示。



注意，如果4200端口由于某种原因被占用了，也可以在其他端口号上启动。仔细阅读你电脑上的输出信息，找出开发服务器的实际URL。

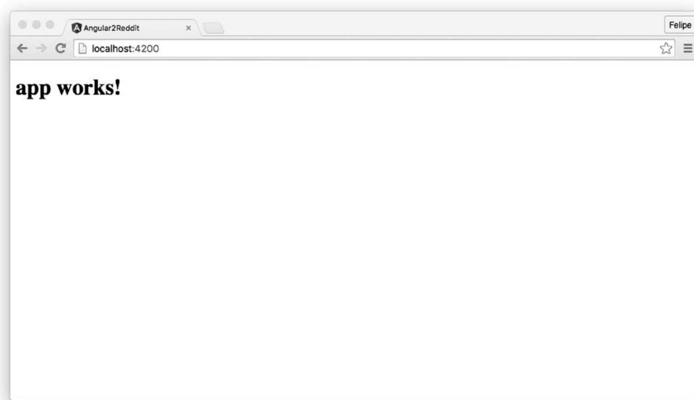


图1-3　运行中的应用

好，现在我们设置好了应用，而且知道了该如何运行它，可以开始写代码了。

1.3.1　制作 Component

Angular背后的指导思想之一就是组件化。

在Angular应用中，我们写HTML标记并把它变成可交互的应用。不过浏览器只认识一部分标签，比如`<select>`、`<form>`和`<video>`等，它们的功能都是由浏览器的开发者预先定义好的。

如果我们想教浏览器认识一些新标签，该怎么办呢？比如我们想要一个`<weather>`标签，用来显示天气；又比如想要一个`<login>`标签，用来创建一个登录面板。

这就是组件化背后的基本思想：我们要教浏览器认识一些拥有自定义功能的新标签。



如果你用过AngularJS，那么可以把组件当作新版本的指令。

让我们来创建第一个组件。写完该组件之后，就能在HTML文档中使用它了，就像这样：

```
<app-hello-world></app-hello-world>
```

要使用angular-cli来创建新组件，可以使用generate（生成）命令。

要生成hello-world组件，我们需要运行下列命令：

```
$ ng generate component hello-world
installing component
  create src/app/hello-world/hello-world.component.css
  create src/app/hello-world/hello-world.component.html
  create src/app/hello-world/hello-world.component.spec.ts
  create src/app/hello-world/hello-world.component.ts
```

那该怎么定义一个新组件呢？最基本的组件包括两部分：

(1) Component注解

(2) 组件定义类

下面来看看组件的代码，然后逐一讲解。打开第一个TypeScript文件：src/app/hello-world/hello-world.component.ts。

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```



注意，TypeScript文件的后缀是.ts而不是.js。问题在于浏览器并不知道该如何解释TypeScript文件。为了解决这个问题，ng serve命令会自动把.ts文件编译为.js文件。

这个代码片段乍一看可能有点恐怖，但别担心，我们接下来就会一步步讲解它。

1.3.2 导入依赖

import语句定义了我们写代码时要用到的那些模块。这里我们导入了两样东西：Component和OnInit。

我们从"@angular/core"模块中导入了组件（import Component）。"@angular/core"部分告诉程序到哪里查找所需的这些依赖。这个例子中，我们告诉编译器："@angular/core"定义并导出了两个JavaScript/TypeScript对象，名字分别是Component和OnInit。

同样，我们还从这个模块中导入了OnInit（import OnInit）。稍后你就会知道，OnInit能帮我们在组件的初始化阶段运行某些代码。不过现在先不用管它。

注意这个import语句的结构是import { things } from wherever格式。我们把{ things }

这部分的写法叫作解构。解构是由ES6和TypeScript提供的一项特性，下一章会深入讲解。

import的用法很像Java中的import或Ruby中的require：从另一个模块中拉取这些依赖，并且让这些依赖在当前文件中可用。

1.3.3　Component 注解

导入依赖后，我们还要声明该组件。

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.ts

```
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
```

如果你习惯用JavaScript编程，那么下面这段代码可能看起来有点怪异：

```
@Component({
  // ...
})
```

这是什么？如果你有Java开发背景，应该会很熟悉：它们是注解。



AngularJS的依赖注入技术在幕后使用了注解的概念。也许你还不熟悉它们，但注解其实是让编译器为代码添加功能的途径之一。

我们可以把注解看作添加到代码上的元数据。当在HelloWorld类上使用@Component时，就把HelloWorld“装饰”（decorate）成了一个Component。

这个<app-hello-world>标签表示我们希望在HTML中使用该组件。要实现它，就得配置@Component并把selector指定为app-hello-world。

```
@Component({
  selector: 'app-hello-world'
  // ... more here
})
```

有很多种方式来配置选择器（selector），类似于CSS选择器、XPath或JQuery选择器。Angular组件对选择器的混用方式添加了一些特有的限制，稍后会谈到。现在，只要记住我们正在定义一个新的HTML标签就可以了。

这里的selector属性用来指出该组件将使用哪个DOM元素。如果模板中有<app-hello-world></app-hello-world>标签，就用该Component类及其组件定义信息对其进行编译。

1.3.4 用 `templateUrl` 添加模板

在这个组件中，我们把`templateUrl`指定为`./hello-world.component.html`。这意味着我们将从与该组件同目录的hello-world.component.html文件中加载模板。下面来看看这个文件。

code/first_app/angular2_hello_world/src/app/hello-world/hello-world.component.html

```
<p>
  hello-world works!
</p>
```

这里定义了一个p标签，其中包含了一些简单的文本。当Angular加载该组件时，就会读取此文件的内容作为组件的模板。

1.3.5 添加 `template`

我们有两种定义模板的方式：使用`@Component`对象中的`template`属性；指定`templateUrl`属性。

我们可以通过传入`template`选项来为`@Component`添加一个模板：

```
@Component({
  selector: 'app-hello-world',
  template: `
    <p>
      hello-world works inline!
    </p>
  `
})
```

注意，我们在反引号中（` ... `）定义了`template`字符串。这是ES6中的一个新特性（而且很棒），允许使用多行字符串。使用反引号定义多行字符串，可以让我们更轻松地把模板放到代码文件中。



你真的应该把模板放进代码文件中吗？答案是：视情况而定。在很长一段时间里，大家都觉得最好把代码和模板分开。这对于一些开发团队来说确实更容易，不过在某些项目中会增加成本，因为你将不得不在一大堆文件之间切换。
个人观点：如果模板行数短于一页，我更倾向于把模板和代码放在一起（也就是.ts文件中）。这样就能同时看到逻辑和视图部分，同时也便于理解它们之间如何互动。
把视图和代码内联在一起的最大缺点是，很多编辑器仍然不支持对内部HTML字符串进行语法高亮。我们期待能尽快看到有更多编辑器支持对模板字符串内嵌HTML的语法高亮。

1.3.6　用 styleUrls 添加 CSS 样式

注意styleUrls属性：

```
styleUrls: ['./hello-world.component.css']
```

这段代码的意思是，我们要使用hello-world.component.css文件中的CSS作为该组件的样式。Angular使用一项叫作样式封装（style-encapsulation）的技术，它意味着在特定组件中指定的样式只会应用于该组件本身。14.1节会深入讨论它。

目前还用不到任何“组件局部样式”，你只要先知道它就行了（或整体删除此属性）。



你可能注意到了该属性与template有个不同点：它接收一个**数组**型参数。这是因为我们可以为同一个组件加载多个样式表。

1.3.7　加载组件

现在，我们已经写完了第一个组件的代码，那该如何把它加载到页面中呢?

如果再次在浏览器中访问此应用，我们会看到一切照旧。这是因为我们仅仅创建了该组件，但还没有使用它。

为了解决这一点，需要把该组件的标签添加到一个将要渲染的模板中去。打开文件first_app/angular2_hello_world/src/app/app.component.html。

记住，因为我们为HelloWorldComponent配置了app-hello-world选择器，所以要在模板中使用<app-hello-world></app-hello-world>。让我们把<app-hello-world>标签添加到app.component.html中。

code/first_app/angular2_hello_world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>
</h1>
```

现在，刷新该页面就会看到如图1-4所示结果。

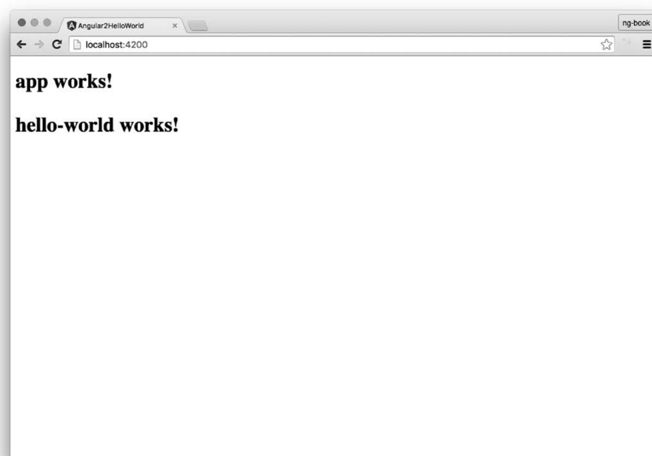


图1-4　“Hello world”一切正常

工作正常！

1.4　把数据添加到组件中

现在，该组件渲染了一个静态模板。这表示我们的组件还不够有趣。

设想有一个应用会显示一个用户列表，并且我们想在其中显示用户的名字。在渲染整个列表之前，需要先渲染一个单独的用户。因此，我们来创建一个新的组件，它将显示用户的名字。

再次使用ng generate命令：

```
ng generate component user-item
```

记住，想看到我们创建好的组件，就要把它添加到一个模板中。

让我们把app-user-item标签添加到app.component.html中，以便看到所作的改动。把app.component.html修改成下面这样。

code/first_app/angular2_hello_world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>

  <app-user-item></app-user-item>
</h1>
```

然后刷新页面，并确认你在该页看到文本user-item works!。

我们希望UserItemComponent显示一个指定用户的名字。

因此，引入name并声明为组件的一个新属性。有了name属性，我们就能在不同的用户之间复用该组件了（但要求页面脚本、逻辑和样式相同）。

为了添加名字，我们要在UserItemComponent类上引入一个属性，来声明该组件有一个名叫name的局部变量。

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

```
export class UserItemComponent implements OnInit {
  name: string; // <-- added name property

  constructor() {
    this.name = 'Felipe'; // set the name
  }

  ngOnInit() {
  }

}
```

注意，我们改变了以下两点。

1. name属性

我们往UserItemComponent类添加了一个属性。注意，这相对于ES5 JavaScript来说是个新语法。在name:string;中，name是我们想设置的属性名，而string是该属性的类型。

为name指定类型是TypeScript中的特性，用来确保它的值必须是string。这些代码在User-ItemComponent类的实例中设置了一个名为name的属性，并且编译器会确保name是一个string。

2. 构造函数

在UserItemComponent类中，我们定义了一个构造函数。这个函数会在创建这个类的实例时自动调用。

在我们的构造函数中，可以通过this.name来设置name属性。

如果这样写：

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

```
  constructor() {
    this.name = 'Felipe'; // set the name
  }
```

就表示当一个新的UserItemComponent组件被创建时，把name设置为'Felipe'。

- **渲染模板**

填好这个值之后，我们可以用模板语法（也就是双花括号语法{{ }}）在模板中显示该变量

的值。

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.html

```
<p>
  Hello {{ name }}
</p>
```

注意，我们在template中引入了一个新的语法：{{ name }}。这些括号叫作“模板标签”（也叫“小胡子标签”）。模板标签中间的任何东西都会被当作一个表达式来展开。这里，因为template是绑定到组件上的，所以name将会被展开为this.name的值，也就是'Felipe'。

3. 试试看

进行这些修改之后，重新加载页面，页面上应该显示Hello Felipe，如图1-5所示。

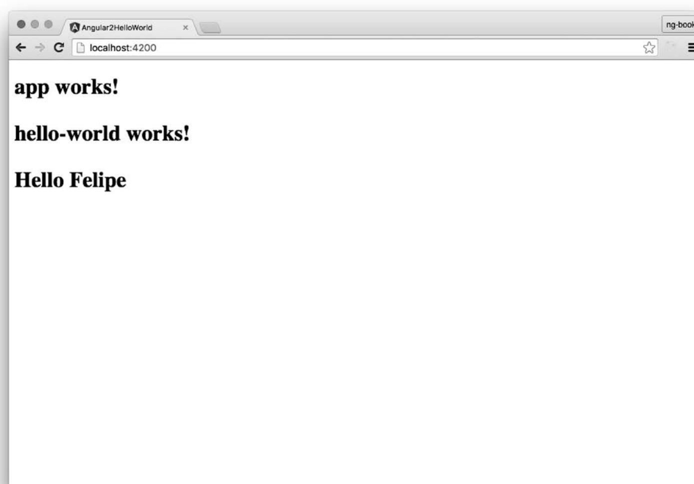


图1-5　带有数据的应用

1.5　使用数组

现在，我们可以对一个单独的名字问好了，但是如果想对一组名字问好呢？

如果你以前用过AngularJS，那么可能用过ng-repeat指令。在Angular中，NgFor是类似的指令（我们在模板标记中通过*ngFor语法来使用它，稍后会讲到）。它们在语法上略有不同，但作用是一样的：为一组对象反复渲染同样的页面脚本。

下面创建一个会渲染用户列表的新组件。我们还是从生成一个新组件开始：

```
ng generate component user-list
```

接着，把app.component.html文件中的<app-user-item>替换为<app-user-list>。

code/first_app/angular2_hello_world/src/app/app.component.html

```
<h1>
  {{title}}

  <app-hello-world></app-hello-world>

  <app-user-list></app-user-list>
</h1>
```

就像给UserItemComponent添加了name属性一样，我们也给UserListComponent添加names属性。

不过，不再设置该属性只存储一个字符串，而是存储一个字符串数组。数组的语法就是在类型后面紧跟一对方括号[]，如下所示。

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.ts

```
export class UserListComponent implements OnInit {
  names: string[];

  constructor() {
    this.names = ['Ari', 'Carlos', 'Felipe', 'Nate'];
  }

  ngOnInit() {
  }

}
```

要留意的第一处变化是在UserListComponent类中添加了一个新的string[]属性。这种语法表示names的类型是string构成的数组。它的另一种写法是Array<string>。

我们还修改了构造函数，让它将this.names的值设置为['Ari', 'Carlos', 'Felipe', 'Nate']。

现在就可以更新模板，渲染出这个名字列表了。这时我们要用到*ngFor，它会在一个列表上进行迭代，为列表中的每一个条目生成一个新标签。新模板如下所示。

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

```
<ul>
  <li *ngFor="let name of names">Hello {{ name }}</li>
</ul>
```

我们用一个ul和一个添加了*ngFor="let name of names"属性的li元素来更新模板。这个*字符和let语法可能会让你摸不着头脑，我们把它们拆开来解释。

*ngFor语法是说我们想在这个属性上使用NgFor指令。你可以把NgFor理解成一个类似于for的循环，其目的是为集合中的每个条目都新建一个DOM元素。

它的值是"let name of names"。names是我们在HelloWorld对象中定义的名字数组。let name叫作引用。"let name of names"的意思是，循环处理names中的每一个元素并将其逐个赋值给一个名叫name的局部变量。

NgFor指令将为数组names中的每一个条目都渲染出一个li标签，并声明一个本地变量name来持有当前迭代的条目。然后，这个新变量将被插值到Hello {{ name }}代码片段里。



并不是必须把这个引用变量命名为name。我们也可以这样写：

```
<li *ngFor="let foobar of names">Hello {{ foobar }}</li>
```

但把顺序反过来行吗？来个小测验吧！如果写成下面这样会如何？

```
<li *ngFor="let name of foobar">Hello {{ name }}</li>
```

当然会出错！因为foobar并不是该组件上的属性。



NgFor会重复渲染ngFor所在的元素。也就是说，我们应该把它放到li标签上而不是ul标签上，因为我们希望重复的是列表元素(li)而非列表本身(ul)。



如果你想进一步探索，可以直接阅读Angular源代码来学习Angular核心团队是如何编写组件的。比如，你能在https://github.com/angular/angular/blob/master/modules/%40angular/common/src/directives/ng_for.ts找到NgFor指令的源代码。

现在刷新页面，就会看到此数组中的每个字符串都有了对应的li，如图1-6所示。

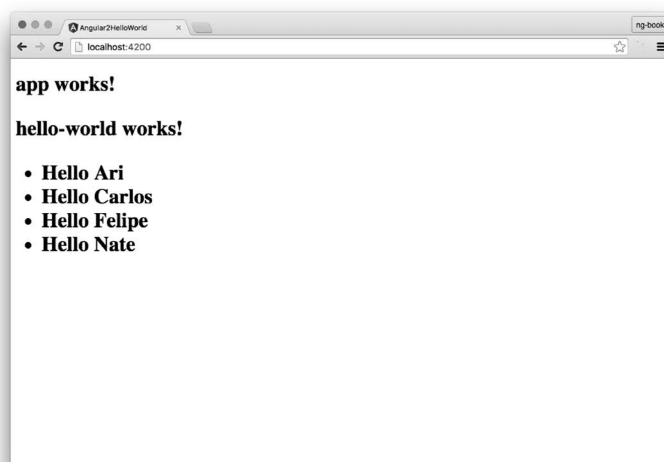


图1-6 带有数据的应用

1.6　使用 UserItemComponent 组件

还记得以前我们创建过UserItemComponent吗？这次不会在UserListComponent中直接渲染每个名字了，而是改用UserItemComponent作为子组件。也就是说，我们不再直接重复渲染li标签，而是让UserItemComponent来为列表中的每个条目指定模板（和功能）。

我们需要做三件事来实现这一点。

(1) 配置UserListComponent来（在它的模板中）渲染UserItemComponent。

(2) 配置UserItemComponent来接收name变量作为输入。

(3) 配置UserListComponent的模板来把用户名传给UserItemComponent。

让我们来逐一完成。

1.6.1　渲染 UserItemComponent

UserItemComponent指定了选择器app-user-item，接下来要把这个标签添加到模板中。我们要做的就是把li标签替换为app-user-item标签。

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

```
<ul>
  <app-user-item
    *ngFor="let name of names">
  </app-user-item>
</ul>
```

注意，当把li标签替换为app-user-item时，我们保留了ngFor属性。这是因为我们仍然要在用户名列表上进行循环。

注意，我们还移除了该模板内部的内容，因为UserItemComponent组件有自己的模板。如果刷新浏览器，看到的结果如图1-7所示。

它确实重复了，但有些不大对劲——每个用户名都是Felipe！我们需要某种方式来把数据传给子组件。

谢天谢地，Angular为此提供了一种方式：@Input注解。

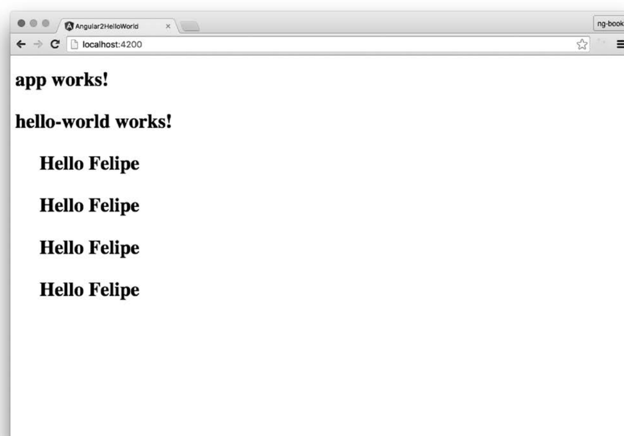


图1-7 带有数据的应用

1.6.2 接收输入

还记得吗？UserItemComponent已经在其构造函数中设置了this.name = 'Felipe';。现在，我们需要进行一些改动，让组件的name属性从外部接收值。

这里要把UserItemComponent修改为：

code/first_app/angular2_hello_world/src/app/user-item/user-item.component.ts

```
import {
  Component,
  OnInit,
  Input // <--- added this
} from '@angular/core';

@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.css']
})
export class UserItemComponent implements OnInit {
  @Input() name: string; // <-- added Input annotation

  constructor() {
    // removed setting name
  }

  ngOnInit() {
  }

}
```

注意，我们修改了name属性，使其具有一个@Input注解。在第3章中，我们会讨论更多关于Input（和Output）的知识，但目前你只要知道该语法能让我们从父模板中传进来一个值就可以了。

为了使用Input，我们还得把它添加到import的列表中去。

最后，我们不希望为name设置默认值，因此从构造函数中移除它。

现在我们有了一个名叫name的Input，那么该如何使用它呢？

1.6.3　传入 Input 值

为了把一个值传入组件，就要在模板中使用方括号[]语法。来看看修改过的模板。

code/first_app/angular2_hello_world/src/app/user-list/user-list.component.html

```
<ul>
  <app-user-item
    *ngFor="let name of names"
    [name]="name">
  </app-user-item>
</ul>
```

注意，我们在app-user-item标签上添加了新属性[name]="name"。在Angular中，添加一个带方括号的属性（比如[foo]）意味着把一个值传给该组件上同名的输入属性（比如foo）。

在这个例子中，name右侧的值来自ngFor中的let name ...语句。也就是说，对于下列代码：

```
  <app-user-item
    *ngFor="let individualUserName of names"
    [name]="individualUserName">
  </app-user-item>
```

[name]部分指定的是UserItemComponent上的Input。注意，我们正在传入的并不是字符串字面量"individualUserName"，而是individualUserName变量的值，也就是names中的每个元素。

在第3章中，我们会详细讲解输入属性和输出属性。现在，你所要知道的是：

(1) 在names中迭代；

(2) 为names中的每个元素创建一个新的UserItemComponent；

(3) 把当前名字的值传给UserItemComponent上名叫name的Input属性。

现在，渲染名字列表的工作就完成了（如图1-8所示）!

恭喜！你已经用组件构建出了你的第一个Angular应用。

当然，该应用非常简单，你应该还希望构建更复杂的应用。别急，在本书中，我们将带你成为编写Angular应用的专家。事实上，我们在本章中还会构建一个投票应用（就像Reddit或Product Hunt）。该应用具有用户交互特性以及更多的组件。

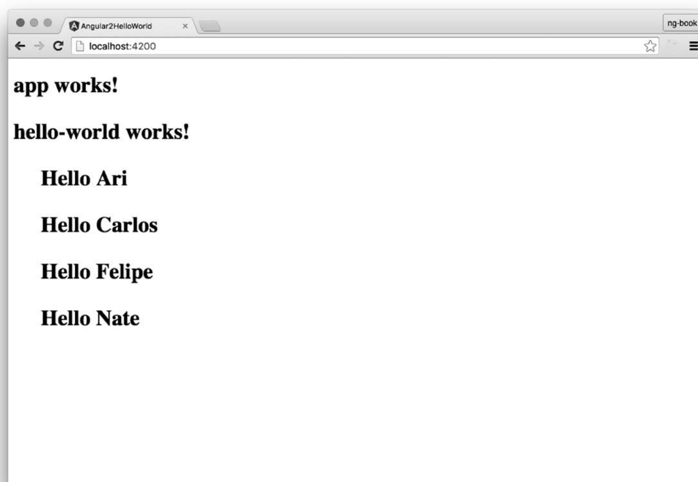


图1-8 应用中的名字一切正常

在开始构建新的应用之前，先来仔细看看Angular应用是如何启动的。

1.7 “启动”速成班

每个应用都有一个主入口点。该应用是由angular-cli构建的，而angular-cli则是基于一个名叫webpack的工具。你不必理解webpack就能使用Angular，但理解应用的启动流程是很有帮助的。

我们可以通过运行下列命令来启动应用：

```
ng serve
```

ng会查阅angular-cli.json文件来找出该应用的入口点。我们来跟踪一下ng是如何找到我们刚刚构建的组件的。

大体来说，过程如下所示：

- angular-cli.json指定一个"main"文件，这里是main.ts；
- main.ts是应用的入口点，并且会引导（bootstrap）我们的应用；
- 引导过程会引导一个Angular模块——我们尚未讨论过模块，不过很快就会谈到；
- 我们使用AppModule来引导该应用，它是在src/app/app.module.ts中指定的；
- AppModule指定了将哪个组件用作顶层组件，这里是AppComponent；
- AppComponent的模板中有一个<app-user-list>标签，它会渲染出我们的用户列表。

我们将在稍后深入讨论这个过程，现在把目光聚焦在Angular的模块系统上：NgModule。

Angular有一个强大的概念：模块。当引导一个Angular应用时，并不是直接引导一个组件，而是创建了一个NgModule，它指向了你要加载的组件。

我们来看看代码。

code/first_app/angular2_hello_world/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    HelloWorldComponent,
    UserItemComponent,
    UserListComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

我们首先看到的是@NgModule注解。像所有注解一样，这段@NgModule(...)代码为紧随其后的AppModule类添加了元数据。

@NgModule注解有三个属性：declarations、imports和bootstrap。

declarations指定了在该模块中定义的组件。你可能已经注意到了，当我们使用ng generate时，它会自动把生成的组件添加到这个列表里！这涉及Angular中的一个重要思想：

要想在模板中使用一个组件，你必须首先在NgModule中声明它。

imports描述了该模块有哪些依赖。我们正在创建一个浏览器应用，因此要导入BrowserModule。

bootstrap告诉Angular，当使用该模块引导应用时，我们要把AppComponent加载为顶层组件。



我们将在8.10节中深入讨论NgModule。

1.8　扩展你的应用

现在我们学会了如何创建一个基本的应用，下面就开始仿造一个Reddit吧。在开始编程之前，你最好先对此应用进行概览并把它拆解为一些逻辑组件，如图1-9所示。

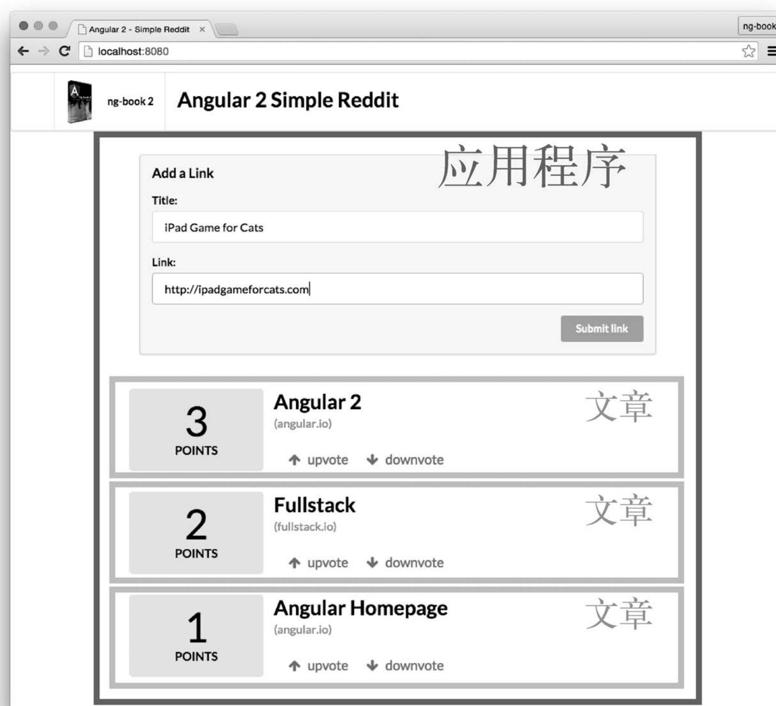


图1-9　应用的逻辑组件

我们将在这个应用程序中构造两个组件：

(1) 整体应用程序，包含一个用来提交新文章的表单（在图1-9中标示为深灰色方框）；

(2) 每个文章（在图1-9中标示为浅灰色方框）。



在较大的应用程序中，用来提交文章的表单本身也应该设计成单独的组件，但是这会让数据传送变得更加复杂。因此为了简化，在本章中我们只使用两个组件。我们目前只创建两个组件，但在本书后面的章节中，我们将学习如何处理更复杂的数据架构。

首先，像以前一样运行ng new命令，并传入一个想要的名字来生成新的应用（这里我们将创建一个名叫angular2_reddit的应用）：

```
ng new angular2_reddit
```



我们在可下载的示例代码中提供了angular2_reddit的完整版。

1.8.1　添加 CSS

我们要做的第一件事是添加一些CSS样式，来让应用不再完全“素颜”。



如果你正在从头构建应用，可以从完成版示例代码的first_app/angular2_reddit目录下复制一些文件过来。

复制以下文件到你的应用目录下：

- src/index.html
- src/styles.css
- src/app/vendor
- src/assets/images

在本项目中，我们将使用Semantic-UI[①]来帮助添加样式。Semantic-UI是一个CSS框架，类似于Zurb Foundation[②]或Twitter Bootstrap[③]。我们的示例代码中已经包含了它，所以你只需要复制上面指定的文件即可。

1.8.2　应用程序组件

现在来构建一个新的组件，它将：

(1) 存储我们的当前文章列表；

(2) 包含一个表单，用来提交新的文章。

我们可以在src/app/app.component.ts文件中找到主应用组件。打开它，可以看到与以前一样的初始内容。

code/first_app/angular2_reddit/src/app/app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

① http://semantic-ui.com/

② http://foundation.zurb.com

③ http://getbootstrap.com

```
  title = 'app works!';
}
```

我们对此模板稍作修改，使其包含一个表单，用于添加链接。我们将从semantic-ui包中借用一点样式来让这个表单看起来更漂亮一些。

code/first_app/angular2_reddit/src/app/app.component.html

```
<form class="ui large form segment">
  <h3 class="ui header">Add a Link</h3>

  <div class="field">
    <label for="title">Title:</label>
    <input name="title">
  </div>
  <div class="field">
    <label for="link">Link:</label>
    <input name="link">
  </div>
</form>
```

我们要创建一个template，它定义了两个input标签：一个用于文章的标题（title），另一个用于文章的链接（link URL）。

刷新浏览器后，你就会看到渲染出了如图1-10所示的表单。

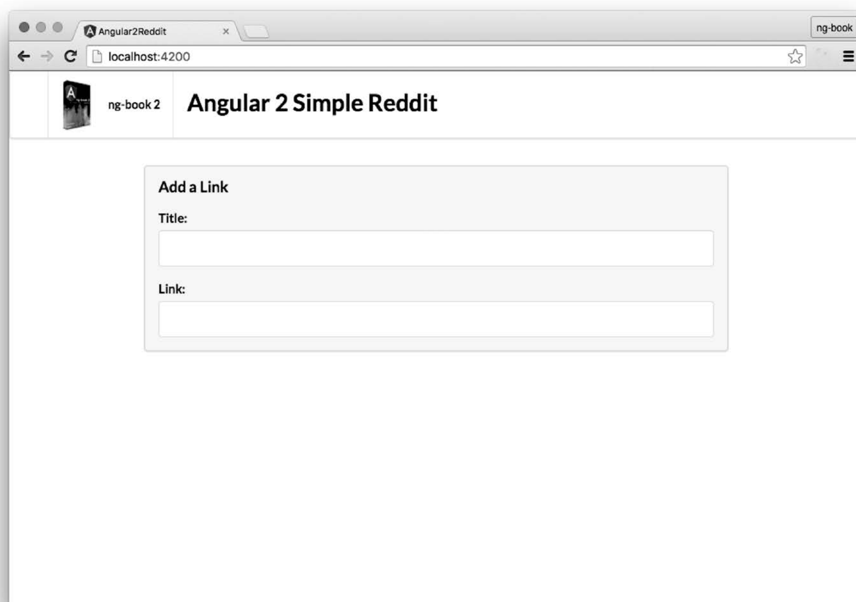


图1-10 表单

1.8.3　添加互动

现在我们有了带input标签的表单，但还没有任何方式来提交数据。下面在表单中添加一个提交按钮，来添加一些交互。

当提交该表单时，我们希望调用一个函数来创建并添加一个链接。可以往<button />元素上添加一个交互事件来实现这个功能。

把事件的名字包裹在圆括号()中就可以告诉Angular：我们要响应这个事件。比如，要想添加一个函数来响应<button />的onClick事件，可以像这样把它传进去：

```
<button (click)="addArticle()"
        class="ui positive right floated button">
  Submit link
</button>
```

这样，当点击这个按钮时，就会调用一个名叫addArticle()的函数；我们要在AppComponent类中定义这个函数。代码如下所示。

code/first_app/angular2_reddit/src/app/app.component.ts

```
export class AppComponent {
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
}
```

一旦把addArticle()函数添加到AppComponent中并且把(click)事件处理器添加到<button />元素上，那么每当点击此按钮时，就会调用该函数。注意，addArticle()函数可以接收两个参数：title和link。我们还要修改模板来把它们传给addArticle()。

我们可以通过为表单中的input元素添加一个特殊的语法来取得模板变量。修改后的模板如下所示。

code/first_app/angular2_reddit/src/app/app.component.html

```
<form class="ui large form segment">
  <h3 class="ui header">Add a Link</h3>

  <div class="field">
    <label for="title">Title:</label>
    <input name="title" #newtitle> <!-- changed -->
  </div>
  <div class="field">
    <label for="link">Link:</label>
    <input name="link" #newlink> <!-- changed -->
  </div>

  <!-- added this button -->
  <button (click)="addArticle(newtitle, newlink)"
```

```
        class="ui positive right floated button">
      Submit link
    </button>

  </form>
```

注意，我们在input标签上使用了#（hash）来要求Angular把该元素赋值给一个局部变量。通过把#title和#link添加到适当的<input />元素上，就可以把它们作为变量传给按钮上的addArticle()函数！

总结一下，我们一共进行了四项修改：

(1) 在模版中创建了一个button标签，向用户表明应该点击哪里；

(2) 新建了一个名叫addArticle的函数，来定义按钮被点击时要做的事情；

(3) 在button上添加了一个(click)属性，意思是“只要点击了这个按钮，就运行addArticle函数”；

(4) 在两个<input>标签上分别添加了#newtitle和#newlink属性。

下面我们按照倒序讲解每一步。

1. 绑定input的值

注意，第一个输入标签是这样的：

```
<input name="title" #newtitle>
```

这段标记告诉Angular把这个<input>绑定到变量newtitle上。#newtitle语法被称作一个解析（resolve），其效果是让变量newtitle可用于该视图的所有表达式中。

newtitle现在是一个对象，它代表了这个input DOM元素（更确切地说，它的类型是HTMLInputElement）。由于newtitle是一个对象，我们可以通过newtitle.value表达式来获取这个输入框的值。

同样，我们把#newlink添加到了另一个<input>标签上，因此也可以用它来提取这个输入框的值。

2. 把事件绑定到动作

我们在button标签上添加了属性(click)来定义点击此按钮时应该怎么做。当发生(click)事件时，我们会调用addArticle并传入两个参数：newtitle和newlink。这个函数和这两个参数是从哪里来的？

(1) addArticle是组件定义类AppComponent里的一个函数。

(2) newtitle来自名叫title的<input>标签上的解析（#newtitle）。

(3) `newlink`来自名叫`link`的`<input>`标签上的解析（`#newlink`）。

全部合并起来是这样的：

```
<button (click)="addArticle(newtitle, newlink)"
        class="ui positive right floated button">
  Submit link
</button>
```



`class="ui positive right floated button"`标签来自Semantic UI，它为这个按钮提供了赏心悦目的绿色。

3. 定义操作逻辑

在`class AppComponent`中，我们定义了一个名叫`addArticle`的新函数。它接收两个参数：`title`和`link`。要注意，`title`和`link`都是`HTMLInputElement`类型的对象，而并非直接输入的值；这一点很重要。要从`input`中获取值，就得调用`title.value`。目前，我们通过`console.log`来输出这些参数。

code/first_app/angular2_reddit/src/app/app.component.ts

```
  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    return false;
  }
```



注意，我们又在使用反引号字符串了。这是ES6中非常便利的一个特性：反引号字符串会展开模板变量！

这里，我们把`${title.value}`放在了字符串中，它最终会被替换成`title.value`的值。

4. 试试看

现在，当你点击提交按钮时，就能看到这条消息被打印到控制台中了（如图1-11所示）。

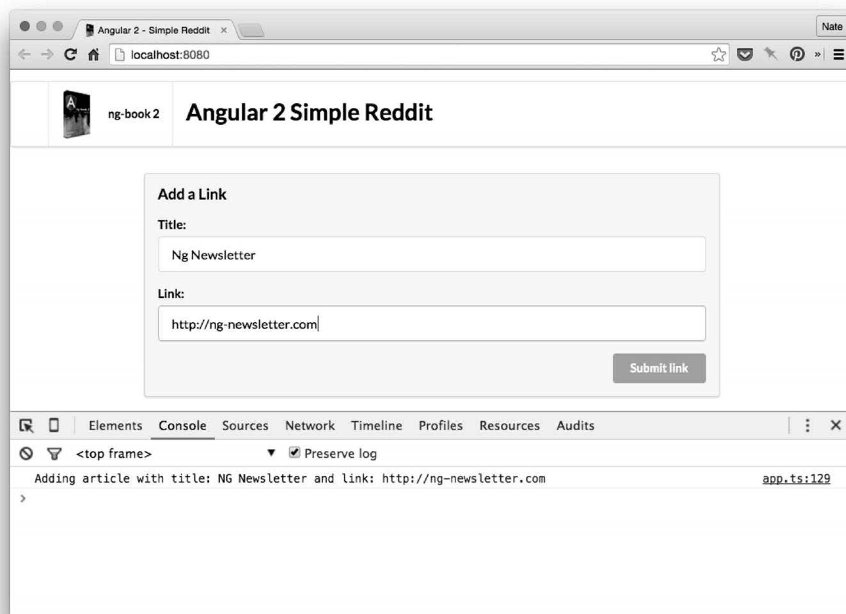


图1-11　点击按钮

1.8.4　添加文章组件

现在，我们有了一个用来提交新文章的表单，但还没有在任何地方展示这些新文章。因为每篇新提交的文章都要显示在本页面的列表中，现在要新建一个组件。

下面就来新建一个组件，用来单独展示这些提交过的文章（如图1-12所示）。

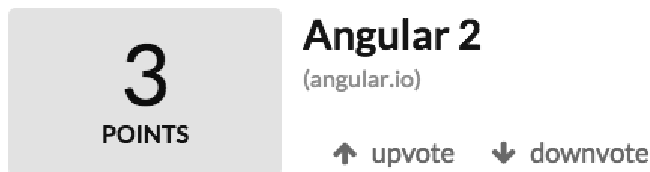


图1-12　一篇文章

为此，我们借助ng工具生成一个新组件：

```
ng generate component article
```

定义这个新组件总共用到了三部分代码：

(1) 在模板中定义了`ArticleComponent`的视图；

(2) 通过为类加上`@Component`注解定义了`ArticleComponent`组件的元数据；

(3) 定义了一个组件定义类（`ArticleComponent`），其中是组件本身的逻辑。

下面来深入讲解一下各部分的细节。

1. 创建`ArticleComponent`的`template`

我们使用文件article.component.html定义模板。

code/first_app/angular2_reddit/src/app/article/article.component.html

```
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value">
      {{ votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ link }}">
    {{ title }}
  </a>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
        <i class="arrow up icon"></i>
          upvote
      </a>
    </li>
    <li class="item">
      <a href (click)="voteDown()">
        <i class="arrow down icon"></i>
          downvote
      </a>
    </li>
  </ul>
</div>
```

这里有很多页面脚本，我们来分解一下（如图1-13所示）。



图1-13　单行文章

我们有两列：

(1) 左侧是投票的数量；

(2) 右侧是文章的信息。

我们分别用`four wide column`和`twelve wide column`这两个CSS类来指定这两列。（记住，它们来自Semantic UI的CSS库。）

我们用模板展开字符串`{{ votes }}`和`{{ title }}`来展示`votes`和`title`。这些值来自`ArticleComponent`类中的`votes`和`title`属性，我们很快就会进行定义。

注意，我们可以在属性值中使用模板字符串，比如在a标签的`href`属性中：`href="{{ link }}"`。在这种情况下，`href`的值会根据组件类的`link`属性的值进行动态插值计算得出。

在upvote和downvote的链接上，我们还有一个动作。只要分别将其按钮上的`(click)`绑定到`voteUp()`和`voteDown()`就可以了。当upvote按钮被按下时，`ArticleComponent`类上的`voteUp()`函数就会被调用；当downvote按钮被按下时，`voteDown()`函数会被调用。

2. 创建ArticleComponent

接下来创建`ArticleComponent`。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css'],
  host: {
    class: 'row'
  }
})
```

首先，我们用`@Component`定义了一个新组件。`selector`表示会用`<app-article>`标签将该组件放在页面中（也就是说，该选择器是一个标签名）。

因此，该组件最基本的使用方式就是把下列标签放在我们的页面脚本中：

```
<app-article>
</app-article>
```

当页面被渲染出来时，这些标签仍然会留在视图中。

我们希望每个`app-article`都独占一行。我们使用的是Semantic UI，它提供了一个用来表示行的CSS类[1]，叫作`row`。

在Angular中，组件的宿主就是该组件所附着到的元素。你会注意到，我们在`@Component`中

① http://semantic-ui.com/collections/grid.html

传入了一个选项：host: { class: 'row' }。它告诉Angular：我们要在宿主元素（app-article标签）上设置class属性，使其具有row类。



这个host选项很不错，它意味着我们可以把app-article的页面脚本封装在组件之内。也就是说，我们不必在使用app-article标签的同时要求父视图中的页面脚本具有class="row"属性。借助host选项，我们就可以在组件的内部配置宿主元素了。

3. 创建组件定义类ArticleComponent

最后，我们来创建组件定义类ArticleComponent。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  votes: number;
  title: string;
  link: string;

  constructor() {
    this.title = 'Angular 2';
    this.link = 'http://angular.io';
    this.votes = 10;
  }

  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }

  ngOnInit() {
  }

}
```

此处我们在ArticleComponent上创建了以下三个属性。

(1) votes：一个数字，用来表示所有“赞”减去所有“踩”的数量之和。

(2) title：一个字符串，用来存放文章的标题。

(3) link：一个字符串，用来存放文章的URL。

在constructor()中，我们设置了一些默认属性。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
  constructor() {
    this.title = 'Angular 2';
    this.link = 'http://angular.io';
    this.votes = 10;
  }
```

我们还为投票定义了两个函数，一个用来“赞”的voteUp和一个用来“踩”的voteDown。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
  voteUp() {
    this.votes += 1;
  }

  voteDown() {
    this.votes -= 1;
  }
```

在voteUp中，我们会把this.votes加一；而在voteDown中，则会把this.votes减一。

4. 使用app-article组件

为了用该组件呈现数据，我们要把<app-article></app-article>标签添加到页面脚本中的某个地方。

这个例子中，我们希望让AppComponent组件来渲染这个新组件。因此修改AppComponent的代码，把<app-article>标签添加到AppComponent的模板中，紧跟在</form>标签后面：

```
  <button (click)="addArticle(newtitle, newlink)"
          class="ui positive right floated button">
    Submit link
  </button>
</form>

<div class="ui grid posts">
  <app-article>
  </app-article>
</div>
```

如果现在刷新浏览器，就会看到<app-article>标签并没有被编译。啊？怎么回事？

无论什么时候遇到这种问题，首先要做的就是打开浏览器的开发者控制台。只要审查一下页面脚本（如图1-14所示），就会看到app-article标签已经出现在页面上了，但是并没有被编译。这是为什么呢？

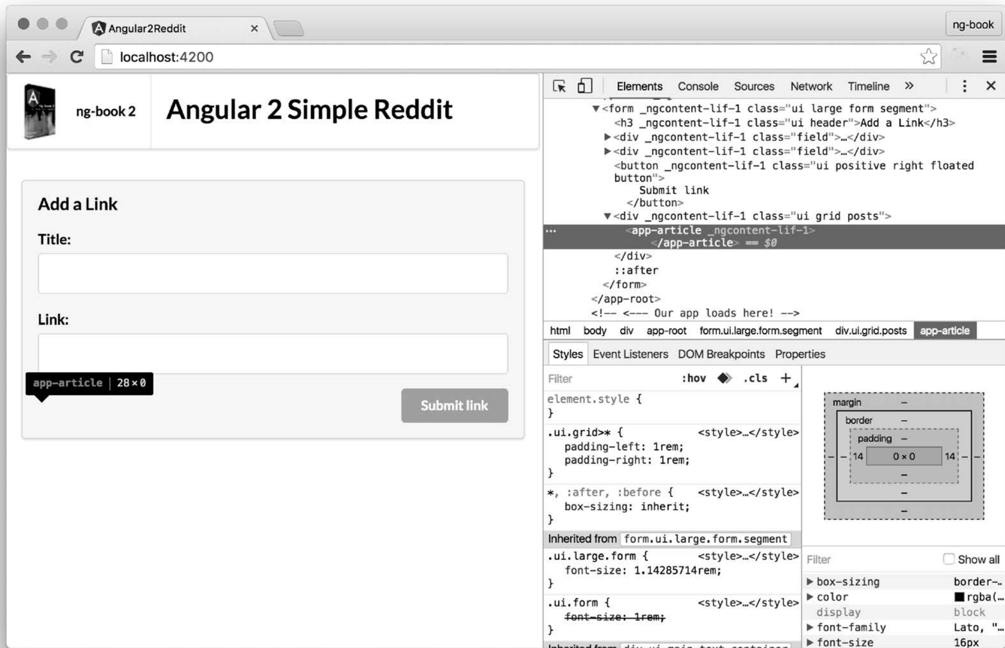


图1-14　审查DOM时未能展开的标记

之所以出现这种情况，是因为AppComponent组件目前还不知道这个ArticleComponent组件。



AngularJS用户注意：如果你用过AngularJS，可能会惊讶于本应用不知道这个新的app-article组件。这是因为在AngularJS中，指令的匹配是全局的；而Angular中，你需要明确指定要使用哪个组件（即哪个选择器）。

一方面，这需要一点配置；但另一方面，这对于构建可伸缩的应用是非常有帮助的，因为这意味着我们不必被迫在全局命名空间中共享这些指令选择器。

为了把这个新的ArticleComponent组件引荐给AppComponent，我们需要把ArticleComponent添加到NgModule的declarations列表中。



之所以要把ArticleComponent添加到declarations中，是因为ArticleComponent是该模块（RedditAppModule）的一部分。然而，如果ArticleComponent是其他模块的一部分，可能就得通过imports来导入它了。

后面还会更深入地讨论NgModule，现在你只需要知道：当创建新组件时，必须同时把它放进NgModule的declarations中。

code/first_app/angular2_reddit/src/app/app.module.ts

```
import { AppComponent } from './app.component';
import { ArticleComponent } from './article/article.component.ts';

@NgModule({
  declarations: [
    AppComponent,
    ArticleComponent // <-- added this
  ],
```

我们在这里：

(1) 用import导入ArticleComponent；

(2) 把ArticleComponent添加到declarations列表中。

把ArticleComponent添加到NgModule的declarations中之后，如果刷新浏览器，就会看到该文章正确渲染出来了（如图1-15所示）。

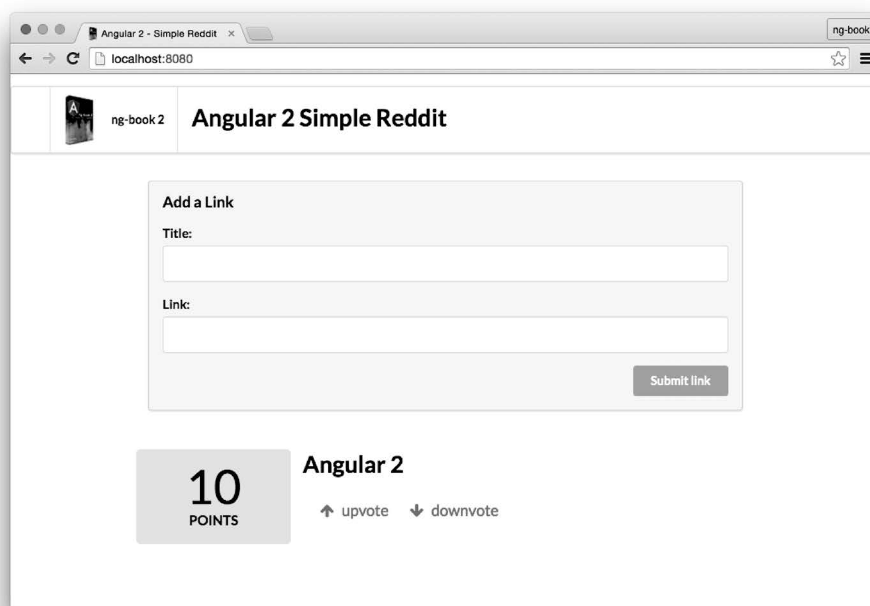


图1-15　渲染ArticleComponent组件

不过，如果你尝试点击“赞”或“踩”的链接，就会看到该页面发生了预料之外的刷新。

在默认情况下，JavaScript会把click事件冒泡到所有父级组件中。因为click事件被冒泡到了父级，浏览器就会尝试导航到这个空白链接，于是浏览器就重新刷新了。

要解决这个问题，我们得让click的事件处理器返回false。这能确保浏览器不会尝试刷新页面。我们要修改代码，以便让每一个voteUp()和voteDown()函数都返回一个布尔值false（告诉浏览器不要向上冒泡）：

```
voteDown(): boolean {
  this.votes -= 1;
  return false;
}
// and similarly with `voteUp()`
```

现在，如果你点击这些链接，就会看到投票数正确地增加或减少了，而且没有出现多余的页面刷新。

1.9　渲染多行

目前，在页面上只有一篇文章，而且也没法渲染更多了，除非我们复制一个<app-article>标签。但即使这样做，所有的文章也都会具有相同的内容，这可不是我们想要的。

1.9.1　创建 **Article** 类

写Angular代码时的最佳实践之一就是尝试从组件代码中把你正在使用的数据结构隔离出来。要做到这一点，就要创建一个数据结构，用以表示单个文章。下面就创建一个新文件article.model.ts来定义所需的Article类吧。

code/first_app/angular2_reddit/src/app/article/article.model.ts

```
export class Article {
  title: string;
  link: string;
  votes: number;

  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }
}
```

此处，我们创建了一个新类，用来表示Article。注意，这是一个普通类而不是Angular组件。在MVC模式中，它被称为模型（model）。

每篇文章都有一个标题title、一个链接link和一个投票总数votes。当创建新文章时，我们需要title和link。votes参数是可选的（用末尾的?标出来），并且默认为0。

现在，我们来修改ArticleComponent的代码，让它使用新的Article类。以前是直接把这些属性存到ArticleComponent组件上，现在则把它改为存到Article类的一个实例上。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular 2',
      'http://angular.io',
      10);
  }

  voteUp(): boolean {
    this.article.votes += 1;
    return false;

  voteDown(): boolean {
    this.article.votes -= 1
    return false;
  }

  ngOnInit() {
  }

}
```

注意我们改动了什么：以前我们直接把title、link和votes属性存到该组件上，而现在则存储一个对article的引用。把article变量的类型设置成新的Article类，代码变整洁了。

接下来修改voteUp（以及voteDown）时，我们不再递增组件上的votes了，而是需要递增article上的votes。

这次重构还引入了另一项修改：我们需要修改视图代码，从正确的位置获取模板变量。这样我们就要修改模板中的标签，使其从article中读取。也就是说，我们以前用的是{{ votes }}，而现在要改成{{ article.votes }}。

code/first_app/angular2_reddit/src/app/article/article.component.html

```
<div class="four wide column center aligned votes">
  <div class="ui statistic">
    <div class="value">
      {{ article.votes }}
    </div>
    <div class="label">
      Points
    </div>
  </div>
</div>
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}">
    {{ article.title }}
  </a>
```

```
    <ul class="ui big horizontal list voters">
      <li class="item">
        <a href (click)="voteUp()">
          <i class="arrow up icon"></i>
            upvote
        </a>
      </li>
      <li class="item">
        <a href (click)="voteDown()">
          <i class="arrow down icon"></i>
            downvote
        </a>
      </li>
    </ul>
  </div>
```

刷新浏览器，仍然一切正常。

情况好多了，但还是有些代码不尽如人意：voteUp和voteDown方法打破了Article类的封装，因为它们直接修改了文章的内部属性。



当前的voteUp和voteDown违反了迪米特法则①。迪米特法则是指：一个对象对其他对象的结构或属性所作的假设应该越少越好。

问题在于ArticleComponent组件了解太多Article类的内部知识了。要解决这一点，就要为Article类添加voteUp和voteDown方法。

code/first_app/angular2_reddit/src/app/article/article.model.ts

```
export class Article {
  title: string;
  link: string;
  votes: number;

  constructor(title: string, link: string, votes?: number) {
    this.title = title;
    this.link = link;
    this.votes = votes || 0;
  }

  voteUp(): void {
    this.votes += 1;
  }

  voteDown(): void {
    this.votes -= 1;
  }

  domain(): string {
```

① http://en.wikipedia.org/wiki/Law_of_Demeter

```
    try {
      const link: string = this.link.split('//')[1];
      return link.split('/')[0];
    } catch (err) {
      return null;
    }
  }
}
```

然后可以修改ArticleComponent组件来调用这些方法。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular',
      'http://angular.io',
      10);
  }

  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }

  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }

  ngOnInit() {
  }

}
```



为什么模型和组件中都有一个voteUp函数？

原因在于，这两个函数所做的事情略有不同。ArticleComponent上的voteUp()函数是与组件的视图有关的，而Article模型上的voteUp()定义了模型上的变化。

也就是说，当投票时，Article类可以对模型上的相应功能进行封装。在真实的应用中，Article模型的内部可能更加复杂，比如向Web服务器发起一个API调用，而你显然不希望这些本属于模型的代码出现在组件的控制器中。

同样，在ArticleComponent中，我们return false;从而“阻止事件冒泡”。这是属于视图的逻辑片段，我们不希望Article模型上的voteUp()函数懂得这些与视图有关的API。也就是说，Article模型应该让投票逻辑从特定的视图中分离出来。

在刷新浏览器之后，仍然一切正常，但我们已经有了更加清晰、更加简单的代码。



查看现在的ArticleComponent组件定义会发现：它太短了！我们把大量逻辑移出组件，放进了模型中。与此对应的MVC指南应该是“胖模型、皮包骨的控制器”[①]；其核心思想是，我们要把大部分领域逻辑移到模型中，以便让组件只做尽可能少的工作。

1.9.2　存储多篇文章

我们再写点代码，展示有多个Article的列表。

从让AppComponent拥有一份文章集合开始。

code/first_app/angular2_reddit/src/app/app.component.ts

```
export class AppComponent {
  articles: Article[];

  constructor() {
    this.articles = [
      new Article('Angular 2', 'http://angular.io', 3),
      new Article('Fullstack', 'http://fullstack.io', 2),
      new Article('Angular Homepage', 'http://angular.io', 1),
    ];
  }

  addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
    console.log(`Adding article title: ${title.value} and link: ${link.value}`);
    this.articles.push(new Article(title.value, link.value, 0));
    title.value = '';
    link.value = '';
    return false;
  }
}
```

注意我们的AppComponent中多了这一行：

```
  articles: Article[];
```

Article[]看起来可能有点陌生。这里的意思是articles是Article的数组。另一种写法是Array<Article>。这种模式被称为泛型。Java、C#和一些别的语言中都有这个概念，意思是你的集合（Array）是有类型的。也就是说，Array是一个集合，它只能存放Article类型的对象。

我们通过在构造函数中设置this.articles来初始化这个数组。

code/first_app/angular2_reddit/src/app/app.component.ts

```
  constructor() {
```

① http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model

```
    this.articles = [
      new Article('Angular 2', 'http://angular.io', 3),
      new Article('Fullstack', 'http://fullstack.io', 2),
      new Article('Angular Homepage', 'http://angular.io', 1),
    ];
  }
```

1.9.3 使用 inputs 配置 ArticleComponent

现在,我们已经有了一个Article模型的列表,该怎么把它们传给ArticleComponent组件呢?

这里我们又用到了Input。以前ArticleComponent类的定义是下面这样的。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  article: Article;

  constructor() {
    this.article = new Article(
      'Angular 2',
      'http://angular.io',
      10);
  }
```

问题的关键是，我们在构造函数中硬编码了一个特定的Article；而制作组件时，不但要能封装，还要能复用。

我们真正想做的是配置要显示的Article。比如,假设我们有article1和article2两篇文章,那就要支持把一个Article型的“参数”传给组件来复用app-article组件，就像这样：

```
<app-article [article]="article1"></app-article>
<app-article [article]="article2"></app-article>
```

Angular通过Component上的Input注解来支持我们这样做：

```
class ArticleComponent {
  @Input() article: Article;
  // ...
```

现在，如果我们有一个Article型的变量myArticle，就可以把它传给视图中的Article-Component了。记住，可以用方括号包裹一个变量来把它传给元素，就像这样：

```
<app-article [article]="myArticle"></app-article>
```

注意这里的语法：我们把输入属性的名字放入方括号中([article]),而该属性的值就是我们要传给此输入属性的那个。

接下来，重点是ArticleComponent实例上的this.article将被设置成myArticle。我们可以把这个过程看作将myArticle变量作为一个参数传给（也就是输入给）了我们的组件。

ArticleComponent组件使用@Input之后变成了下面这样。

code/first_app/angular2_reddit/src/app/article/article.component.ts

```
export class ArticleComponent implements OnInit {
  @Input() article: Article;

  voteUp(): boolean {
    this.article.voteUp();
    return false;
  }

  voteDown(): boolean {
    this.article.voteDown();
    return false;
  }

  ngOnInit() {
  }

}
```

1.9.4　渲染文章列表

我们之前配置过AppComponent来存储articles数组。这次我们要配置AppComponent来渲染所有articles。要实现这个功能，就不能单独使用<app-article>标签了，而要用NgFor指令在articles数组上进行迭代，并为其中的每一个都渲染一份app-article。

把下列内容添加到AppComponent前面@Component注解的template属性中，紧跟着</form>标签：

```
    Submit link
  </button>
</form>

<!-- start adding here -->
<div class="ui grid posts">
  <app-article
    *ngFor="let article of articles"
    [article]="article">
  </app-article>
</div>
<!-- end adding here -->
```

还记得我们之前用过NgFor指令把名称列表渲染成无序列表吗？它在渲染多个组件时也同样适用。

*ngFor="let article of articles"语法会对articles列表进行迭代，并且为列表中的每一个条目创建一个局部变量article。

要为组件指定一个输入属性article，就要使用[inputName]="inputValue"表达式。在这个

例子中，该表达式的意思是：我们要把输入属性article设置为局部变量article的值，而后者是由ngFor所设置的。



article变量在这个代码片段中出现的次数太多了。如果我们把NgFor创建的临时变量命名为foobar，或许更清楚一些：

```
<app-article
  *ngFor="let foobar of articles"
  [article]="foobar">
</app-article>
```

那么，这里就有了三个变量：

(1) articles是一个Article的数组，由AppComponent组件定义；

(2) foobar是一个articles数组中的单个元素（一个Article对象），由NgFor定义；

(3) article是一个字段名，由ArticleComponent中的inputs属性定义。

本质上，NgFor首先生成了一个临时变量foobar，然后我们把它传给了app-article。

刷新浏览器，就会看到所有的文章都渲染出来了（如图1-16所示）。

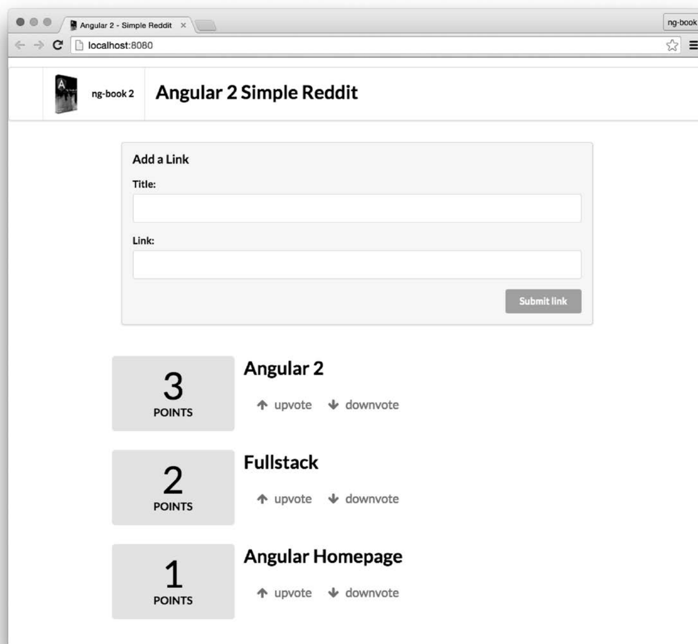


图1-16　渲染多篇文章

1.10　添加新文章

现在，我们需要修改addArticle以便在按下按钮时实际添加一篇新文章。修改addArticle方法，使其变成下面这样。

code/first_app/angular2_reddit/src/app/app.component.ts

```
addArticle(title: HTMLInputElement, link: HTMLInputElement): boolean {
  console.log(`Adding article title: ${title.value} and link: ${link.value}`);
  this.articles.push(new Article(title.value, link.value, 0));
  title.value = '';
  link.value = '';
  return false;
}
```

这将会：

(1) 创建一个具有所提交标题和URL的Article新实例；

(2) 把它加入Article数组；

(3) 清除input字段的值。



我们要如何清除input字段的值呢？回忆一下，title和link都是HTMLInputElement对象。这就意味着我们可以设置它们的属性。当我们修改value属性时，页面中的input标签也会跟着改变。

在输入框中添加新文章，并点击Submit Link之后，就会看到新的文章添加成功了！

1.11　最后的修整

1.11.1　显示文章所属的域名

我们先为链接添加一个提示信息，以便在用户点击链接时显示将重定向到的域名。

把domain方法添加到Article类中。

code/first_app/angular2_reddit/src/app/article/article.model.ts

```
domain(): string {
  try {
    const link: string = this.link.split('//')[1];
    return link.split('/')[0];
  } catch (err) {
    return null;
  }
}
```

把对该函数的调用添加到ArticleComponent的模板中：

```
<div class="twelve wide column">
  <a class="ui large header" href="{{ article.link }}">
    {{ article.title }}
  </a>
  <!-- right here -->
  <div class="meta">({{ article.domain() }})</div>
  <ul class="ui big horizontal list voters">
    <li class="item">
      <a href (click)="voteUp()">
```

现在，当我们刷新浏览器时，就能看到每个URL所属的域名了（注意：URL必须包含http://）。

1.11.2 基于分数重新排序

如果你点击并投票，就会发现有些事情不太对劲：这些文章并没有基于分数排序！显然，我们更希望让分数最高的条目显示在顶部，让低分条目沉到底部。

我们把articles存储在了AppComponent类中，但这个数组是无序的。处理这种情况的简单方式是在AppComponent上创建一个新方法sortedArticles。

code/first_app/angular2_reddit/src/app/app.component.ts

```
  sortedArticles(): Article[] {
    return this.articles.sort((a: Article, b: Article) => b.votes - a.votes);
  }
```

这样，在ngFor中，我们就可以在sortedArticles()上而不是直接在articles上迭代了：

```
<div class="ui grid posts">
  <app-article
    *ngFor="let article of sortedArticles()"
    [article]="article">
  </app-article>
</div>
```

1.12 全部代码

在本章中，我们浏览了代码中的很多小片段。你可以到本书示例代码的下载站点找到该应用的全部文件和完整的TypeScript代码。

1.13 总结

完工！我们已经创建了自己的第一个Angular应用。还不错，对吧？不过我们还会学到更多：理解数据流、发起AJAX请求、内置指令、路由、操纵DOM，等等。

现在，好好享受成功的喜悦吧！很多Angular程序的写法都和我们刚刚所做的类似：

(1) 把应用拆分成组件；

(2) 创建视图；

(3) 定义模型；

(4) 显示模型；

(5) 添加交互。

在后面的章节中，我们将讲解用Angular编写各种复杂应用的全部知识。

1.14　获得帮助

如果你有关于本章的任何问题，比如发现了bug或在运行代码时遇到问题，欢迎告诉我们！

- （英文）加入我们的免费社区，在Gitter上跟我们聊聊：https://gitter.im/ng-book/ng-book。
- （英文）直接给我们发送邮件：us@fullstack.io。
- （中文）如果是与中文版相关的问题与勘误，请访问我们的GitHub：https://github.com/ng-book2/book。
- （中文）获取官方文档中文版，请访问angular.cn。
- （中文）如果想了解本书范围之外的问题，请访问wx.angular.cn向我们提问。
- （中文）要了解Angular的最新消息，欢迎搜索并关注微信公众号：Angular中文社区。

继续前进吧！

第 2 章 TypeScript

2.1 Angular 是用 TypeScript 构建的

Angular是用一种类似于JavaScript的语言——TypeScript[①]——构建的。

或许你会对用新语言来开发Angular心存疑虑，但事实上，在开发Angular应用时，我们有充分的理由用TypeScript代替普通的JavaScript。

TypeScript并不是一门全新的语言，而是ES6的超集。所有的ES6代码都是完全有效且可编译的TypeScript代码。图2-1展示了它们之间的关系。

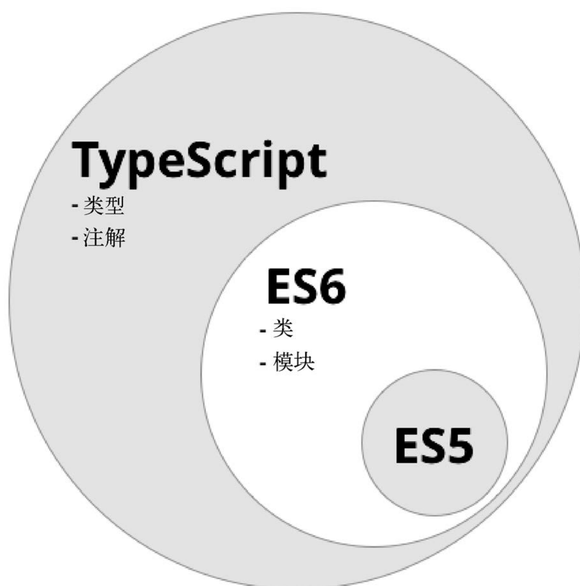


图2-1　ES5、ES6和TypeScript

① http://www.typescriptlang.org/



什么是ES5？什么是ES6？ES5是ECMAScript 5的缩写，也被称为“普通的JavaScript”。ES5就是大家熟知的JavaScript，它能够运行在大部分浏览器上。ES6则是下一个版本的JavaScript，在后续章节中我们还会深入讨论它。

在本书出版的时候，支持ES6的浏览器还很少，更不用说TypeScript了。我们用转译器来解决这个问题。TypeScript转译器能把TypeScript代码转换为几乎所有浏览器都支持的ES5代码。



从TypeScript代码到ES5代码的唯一转换器是由TypeScript核心团队编写的。然而，将ES6代码（不是TypeScript代码）转换到ES5代码则有两个主要的转换器：Google开发的Traceur[①]与JavaScript社区创建的Babel[②]。在本书中我们并不会直接使用它们，但它们也是值得了解的不错项目。

我们在上一章安装了TypeScript环境，如果你是从本章开始学习的，那么可以这样安装TypeScript环境：`npm install -g typescript`。

TypeScript是Microsoft和Google之间的官方合作项目。有这两家强有力的科技巨头在背后支撑，对于我们来说是个好消息，因为这表示TypeScript将会得到长期的支持。这两家公司都承诺全力推动Web技术的发展，我们这些开发人员显然会获益匪浅。

另外，转译器的好处还在于：它允许小型团队对语言进行改善，而不必要求所有人都去升级他们的浏览器。

需要指出的是：TypeScript并不是开发Angular应用的必选语言。我们同样可以使用ES5代码（即“普通”JavaScript）来开发Angular应用。Angular也为全部功能提供了ES5 API。那么为什么我们还要使用TypeScript呢？这是因为TypeScript有不少强大的功能，能极大地简化开发。

2.2　TypeScript提供了哪些特性

TypeScript相对于ES5有五大改善：

- 类型
- 类
- 注解
- 模块导入
- 语言工具包（比如，解构）

接下来我们逐个介绍。

---

① https://github.com/google/traceur-compiler

② https://babeljs.io/

2.3　类型

顾名思义，相对于ES6，TypeScript最大的改善是增加了类型系统。

有些人可能会觉得，缺乏类型检查正是JavaScript这些弱类型语言的优点。也许你对类型检查心存疑虑，但我仍然鼓励你试一试。类型检查的好处有：

(1) 有助于代码的编写，因为它可以在编译期预防bug；

(2) 有助于代码的阅读，因为它能清晰地表明你的意图。

另外值得一提的是，TypeScript中的类型是可选的。如果希望写一些快速代码或功能原型，可以首先省略类型，然后再随着代码日趋成熟逐渐加上类型。

TypeScript的基本类型与我们平时所写JavaScript代码中用的隐式类型一样，包括字符串、数字、布尔值等。

直到ES5，我们都在用var关键字定义变量，比如var name;。

TypeScript的新语法是从ES5自然演化而来的，仍沿用var来定义变量，但现在可以同时为变量名提供可选的变量类型了：

```
var name: string;
```

在声明函数时，也可以为函数参数和返回值指定类型：

```
function greetText(name: string): string {
  return "Hello " + name;
}
```

这个例子中，我们定义了一个名为greetText的新函数，它接收一个名为name的参数。name: string语法表示函数想要的name参数是string类型。如果给该函数传一个string以外的参数，代码将无法编译通过。对我们来说，这是好事，否则这段代码将会引入bug。

或许你还注意到了，greetText函数在括号后面还有一个新语法:string {。冒号之后指定的是该函数的返回值类型，在本例中为string。这很有用，原因有二：如果不小心让函数返回了一个非string型的返回值，编译器就会告诉我们这里有错误；使用该函数的开发人员也能很清晰地知道自己将会拿到什么类型的数据。

我们来看看如果写了不符合类型声明的代码会怎样：

```
function hello(name: string): string {
    return 12;
}
```

当尝试编译代码时，将会得到下列错误：

```
$ tsc compile-error.ts
compile-error.ts(2,12): error TS2322: Type 'number' is not assignable to type 'string'.
```

这是怎么回事？我们尝试返回一个number类型的12，但hello函数期望的返回值类型为string（它是在参数声明的后面以): string {的形式声明的）。

要纠正它，可以把函数的返回值类型改为number：

```
function hello(name: string): number {
    return 12;
}
```

虽然这只是一个小例子，但足以证明类型检查能为我们节省大量调试bug的时间。

现在知道了如何使用类型，但怎么才能知道有哪些可用类型呢？接下来我们就会罗列出这些内置的类型，并教你如何创建自己的类型。

尝试 REPL

为了运行本章中的例子，我们要先安装一个小工具，名为TSUN[①]（TypeScript Upgraded Node，支持TypeScript的升级版Node）：

```
$ npm install -g tsun
```

接着启动它：

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl

>
```

这个小小的>是一个命令提示符，表示TSUN已经准备好接收命令了。

对于本章后面的大部分例子，你都可以复制粘贴到这个终端窗口中运行。

2.4　内置类型

2.4.1　字符串

字符串包含文本，声明为string类型：

```
var name: string = 'Felipe';
```

2.4.2　数字

无论整数还是浮点，任何类型的数字都属于number类型。在TypeScript中，所有的数字都是

① https://github.com/HerringtonDarkholme/typescript-repl

用浮点数表示的，这些数字的类型就是number：

```
var age: number = 36;
```


2.4.3 布尔类型

布尔类型（boolean）以true（真）和false（假）为值。

```
var married: boolean = true;
```

2.4.4 数组

数组用Array类型表示。然而，因为数组是一组相同数据类型的集合，所以我们还需要为数组中的条目指定一个类型。

我们可以用Array<type>或者type[]语法来为数组条目指定元素类型：

```
var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

数字型数组的声明与之类似：

```
var jobs: Array<number> = [1, 2, 3];
var jobs: number[] = [4, 5, 6];
```

2.4.5 枚举

枚举是一组可命名数值的集合。比如，如果我们想拿到某人的一系列角色，可以这么写：

```
enum Role {Employee, Manager, Admin};
var role: Role = Role.Employee;
```

默认情况下，枚举类型的初始值是0。我们也可以调整初始化值的范围：

```
enum Role {Employee = 3, Manager, Admin};
var role: Role = Role.Employee;
```

在上面的代码中，Employee的初始值被设置为3而不是0。枚举中其他项的值是依次递增的，意味着Manager的值为4，Admin的值为5。同样，我们也可以单独为枚举中的每一项指定值：

```
enum Role {Employee = 3, Manager = 5, Admin = 7};
var role: Role = Role.Employee;
```

还可以从枚举的值来反查它的名称：

```
enum Role {Employee, Manager, Admin};
console.log('Roles: ', Role[0], ',', Role[1], 'and', Role[2]);
```

2.4.6　任意类型

如果我们没有为变量指定类型，那它的默认类型就是any。在TypeScript中，any类型的变量能够接收任意类型的数据：

```
var something: any = 'as string';
something = 1;
something = [1, 2, 3];
```

2.4.7　“无”类型

void意味着我们不期望那里有类型。它通常用作函数的返回值，表示没有任何返回值：

```
function setName(name: string): void {
  this.name = name;
}
```

2.5　类

JavaScript ES5采用的是基于原型的面向对象设计。这种设计模型不使用类，而是依赖于*原型*。

JavaScript社区采纳了大量最佳实践，以弥补JavaScript缺少类的问题。这些最佳实践已经被总结在Mozilla的开发指南中了[①]，你还可以找到一篇关于JavaScript面向对象设计的优秀概述[②]。

不过，在ES6中，我们终于有内置的类了。

用class关键字来定义一个类，紧随其后的是类名和类的代码块：

```
class Vehicle {
}
```

类可以包含*属性*、*方法*以及*构造函数*。

2.5.1　属性

属性定义了类实例对象的数据。比如名叫Person的类可能有first_name、last_name和age属性。

类中的每个属性都可以包含一个可选的类型。比如，我们可以把first_name和last_name声明为字符串类型（string），把age声明为数字类型（number）。

Person类的声明是这样的：

① https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
② https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

```
class Person {
  first_name: string;
  last_name: string;
  age: number;
}
```


2.5.2 方法

方法是运行在类对象实例上下文中的函数。在调用对象的方法之前，必须要有这个对象的实例。



要实例化一个类，我们使用new关键字。比如new Person()会创建一个Person类的实例对象。

如果我们希望问候某个Person，就可以这样写：

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  greet() {
     console.log("Hello", this.first_name);
  }
}
```

注意，借助this关键字，我们能用this.first_name表达式来访问Person类的first_name属性。

如果没有显式声明过方法的返回类型和返回值，就会假定它可能返回任何东西（即any类型）。然而，因为这里没有任何显式的return语句，所以实际返回的类型是void。



注意，void类型也是一种合法的any类型。

调用greet方法之前，我们要有一个Person类的实例对象。代码如下：

```
// declare a variable of type Person
var p: Person;

// instantiate a new Person instance
p = new Person();

// give it a first_name
p.first_name = 'Felipe';

// call the greet method
p.greet();
```



我们还可以将对象的声明和实例化缩写为一行代码：
var p: Person = new Person();

假设我们希望Person类有一个带返回值的方法。比如，要获取某个Person在数年后的年龄，我们可以这样写：

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}

// instantiate a new Person instance
var p: Person = new Person();

// set initial age
p.age = 6;

// how old will he be in 12 years?
p.ageInYears(12);

// -> 18
```

2.5.3　构造函数

构造函数是当类进行实例化时执行的特殊函数。通常会在构造函数中对新对象进行初始化工作。

构造函数必须命名为constructor。因为构造函数是在类被实例化时调用的，所以它们可以有输入参数，但不能有任何返回值。



我们要通过调用new ClassName()来执行构造函数，以完成类的实例化。

当类没有显式地定义构造函数时，将自动创建一个无参构造函数：

```
class Vehicle {
}
var v = new Vehicle();
```

它等价于：

```
class Vehicle {
  constructor() {
  }
}
var v = new Vehicle();
```



在TypeScript中，每个类只能有一个构造函数。
这是违背ES6标准的。在ES6中，一个类可以拥有不同参数数量的多个构造函数重载实现。

我们可以使用带参数的构造函数来将对象的创建工作参数化。

比如，我们可以对Person类使用构造函数来初始化它的数据：

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  constructor(first_name: string, last_name: string, age: number) {
    this.first_name = first_name;
    this.last_name = last_name;
    this.age = age;
  }

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}
```


用下面这种方法重写前面的例子要容易些：

```
var p: Person = new Person('Felipe', 'Coury', 36);
p.greet();
```

当创建这个对象的时候，其姓名、年龄都会被初始化。

2.5.4 继承

面向对象的另一个重要特性就是继承。继承表明子类能够从父类得到它的行为。然后，我们就可以在这个子类中重写、修改以及添加行为。



如果要深入了解ES5的继承是如何工作的，可以参考Mozilla开发文档中的文章“Inheritance and the prototype chain”[1]。

TypeScript是完全支持继承特性的，并不像ES5那样要靠原型链实现。继承是TypeScript的核心语法，用`extends`关键字实现。

要说明这一点，我们来创建一个`Report`类：

```
class Report {
  data: Array<string>;

  constructor(data: Array<string>) {
    this.data = data;
  }

  run() {
    this.data.forEach(function(line) { console.log(line); });
  }
}
```

这个`Report`类有一个字符串数组类型的`data`的属性。当我们调用`run`方法时，它会循环这个`data`数组中的每一项数据，然后用`console.log`打印出来。



`.forEach`是`Array`中的一个方法，它接收一个函数作为参数，并对数组中的每一个条目逐个调用该函数。

给`Report`增加几行数据，并调用`run`把这些数据打印到控制台：

```
var r: Report = new Report(['First line', 'Second line']);
r.run();
```

运行结果如下：

```
First line
Second line
```

现在，假设我们希望有第二个报表，它需要增加一些头信息和数据，但我们仍想复用现有`Report`类的`run`方法来向用户展示数据。

为了复用`Report`类的行为，要使用`extends`关键字来继承它：

```
class TabbedReport extends Report {
  headers: Array<string>;

  constructor(headers: string[], values: string[]) {
    super(values)
```

① https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

```
      this.headers = headers;
    }

    run() {
      console.log(this.headers);
      super.run();
    }
  }

var headers: string[] = ['Name'];
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
var r: TabbedReport = new TabbedReport(headers, data)
r.run();
```

2.6　工具

ES6和TypeScript提供了许多语法特性，让编码成为一种享受。其中最重要的两点是：

- 胖箭头函数语法
- 模板字符串

2.6.1　胖箭头函数

胖箭头（=>）函数是一种快速书写函数的简洁语法。

在ES5中，每当我们要用函数作为方法参数时，都必须用function关键字和紧随其后的花括号（{}）表示。就像这样：

```
// ES5-like example
var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });
```

现在我们可以用=>语法来重写它了：

```
// Typescript example
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach( (line) => console.log(line) );
```

当只有一个参数时，圆括号可以省略。箭头（=>）语法可以用作表达式：

```
var evens = [2,4,6,8];
var odds = evens.map(v => v + 1);
```

也可以用作语句：

```
data.forEach( line => {
  console.log(line.toUpperCase())
});
```

=>语法还有一个重要的特性，就是它和环绕它的外部代码共享同一个this。这是它和普通

function写法最重要的不同点。通常，我们用function声明的函数有它自己的this。有时在JavaScript中能看见如下代码：

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    var self = this;
    this.guitars.forEach(function(g) {
      // this.name is undefined so we have to use self.name
      console.log(self.name + " plays a " + g);
    });
  }
};
```

由于胖箭头会共享环绕它的外部代码的this，我们可以这样改写：

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    this.guitars.forEach( (g) => {
      console.log(this.name + " plays a " + g);
    });
  }
};
```

可见，箭头函数是处理内联函数的好办法。这也让我们在JavaScript中更容易使用高阶函数。

2.6.2　模板字符串

ES6引入了新的模板字符串语法，它有两大优势：

(1) 可以在模板字符串中使用变量（不必被迫使用+来拼接字符串）；

(2) 支持多行字符串。

1. 字符串中的变量

这种特性也叫字符串插值（string interpolation）。你可以在字符串中插入变量，做法如下：

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

注意，字符串插值必须使用反引号，不能用单引号或双引号。

2. 多行字符串

反引号字符串的另一个优点是允许多行文本：

```
var template = `
<div>
  <h1>Hello</h1>
  <p>This is a great website</p>
</div>
`

// do something with `template`
```

当我们要插入模板这样的长文本字符串时，多行字符串会非常有帮助。

2.7 总结

在TypeScript和ES6中还有很多其他的优秀语法特性，如：

- ❑ 接口
- ❑ 泛型
- ❑ 模块的导入、导出
- ❑ 标注
- ❑ 解构

我们会在本书的后续章节中讲到这些概念并使用它们。目前，本章的这些基本知识已经足够你开始学习Angular了。

言归正传，让我们回到Angular吧！

第 3 章

Angular的工作原理

本章将讨论Angular中的高级概念，从全局视角来分析各细节部分是如何协同工作的。



如果你用过AngularJS，会发现Angular采用了全新的思维模型来构建应用。别担心，作为AngularJS的使用者，我们觉得Angular的设计既简明又熟悉。在本书稍后的章节中，我们会专门讨论如何将AngularJS应用转换成Angular应用。

在后面的章节里，我们会对每一个概念进行深入讲解，但目前只作概述并解释最基础的概念。

第一个重要概念：Angular应用是由组件构成的。可以将组件理解为一种教浏览器认识新HTML标签的方式。如果你有使用AngularJS的经验，那么可以把组件理解为类似于指令的概念。（事实上，Angular中也有指令，我们会在后面讨论具体的差异。）

其实，相比AngularJS中的指令，Angular中的组件有一些重要优势，我们会详细讨论。现在，让我们先来看看最顶级的概念：应用。

3.1 应用

一个Angular应用其实就是一棵由组件构成的树。

在这棵树的根结点，最顶层的组件就是应用本身。它会在浏览器启动（也叫引导）应用的时候被渲染。

组件有一个很棒的特性，那就是它们是可组合的。这意味着我们可以基于小组件构建大组件。应用只是一个会渲染其他组件的组件而已。

由于组件是以树型结构组织起来的，当每个组件被渲染时，它都会递归地渲染下级组件。

举个例子，让我们基于如图3-1所示的原型图创建一个简单的库存管理系统。

拿到这个原型图后，我们应该做的第一件事就是把页面拆分成组件。

在这个例子里，我们可以对页面内容进行分组，并抽象成三个高层级组件：

(1) 主导航组件

(2) 面包屑导航组件

(3) 产品列表组件

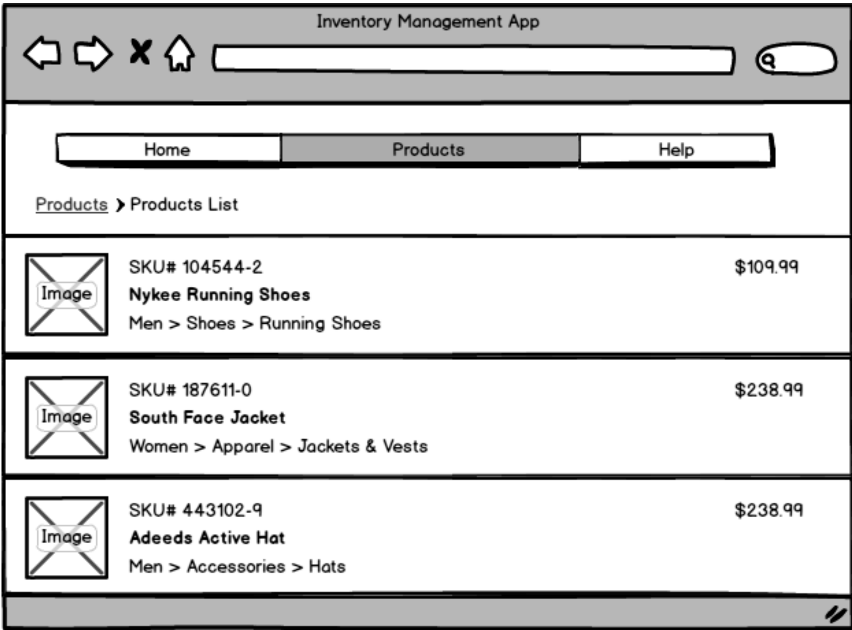


图3-1　库存管理系统

3.1.1　主导航组件

这个组件用来展示主导航部分，用户可以通过主导航组件访问应用的其他部分（如图3-2所示）。



图3-2　主导航组件

3.1.2　面包屑导航组件

这个组件用来展示用户在本应用“网站地图”中的当前位置（如图3-3所示）。



图3-3　面包屑导航组件

3.1.3　产品列表组件

产品列表组件用来展示一组产品（如图3-4所示）。

图3-4　产品列表组件

我们还可以继续拆分产品列表组件，从而得到下一级的产品条目组件（如图3-5所示）。

图3-5　产品条目组件

当然，我们可以再进一步，把每个产品条目组件拆分为更小的组件。

- 产品图片组件用来根据指定的图片名称显示产品图片。
- 产品分类组件用来展示产品分类树。比如：男装 > 鞋 > 跑鞋。
- 价格显示组件用来展示产品价格。如果我们对产品价格有定制化需求，比如用户登录后可以获得全局折扣或者包邮，就可以在这个组件中实现。

最后，把以上组件按层级结构进行整理，就得到了如图3-6所示的树状图。

在树状图的顶层可以看到我们的应用：*库存管理系统*。

往下细分为主导航、面包屑导航和产品列表组件。

产品列表组件包含一些产品条目组件，每个产品各一个。

产品条目组件又包含三个更下层的组件：一个用于展示图片，一个用于展示分类，一个用于展示价格。

现在，让我们一起来实现这个应用。



你可以在本书下载内容的how_angular_works/inventory_app目录中找到本章涉及的全部代码。

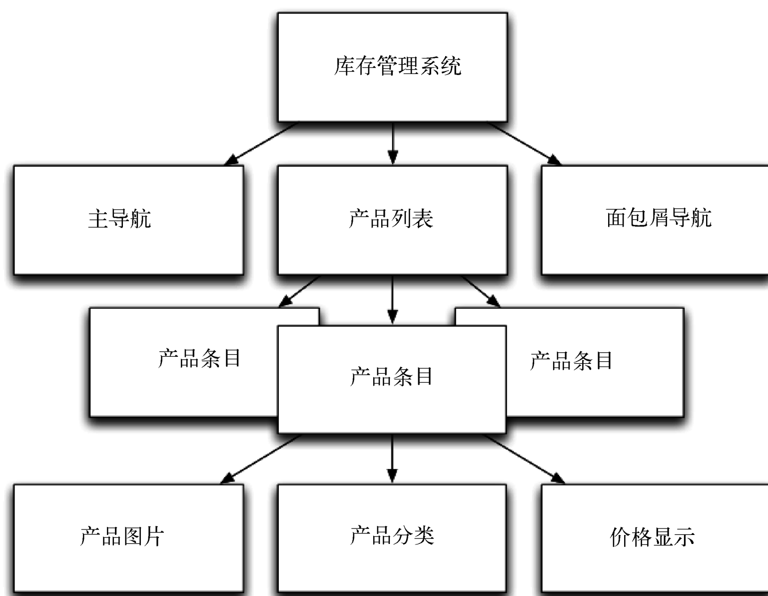


图3-6　应用树状图

当我们的应用完成之后，它看起来应该如图3-7所示。

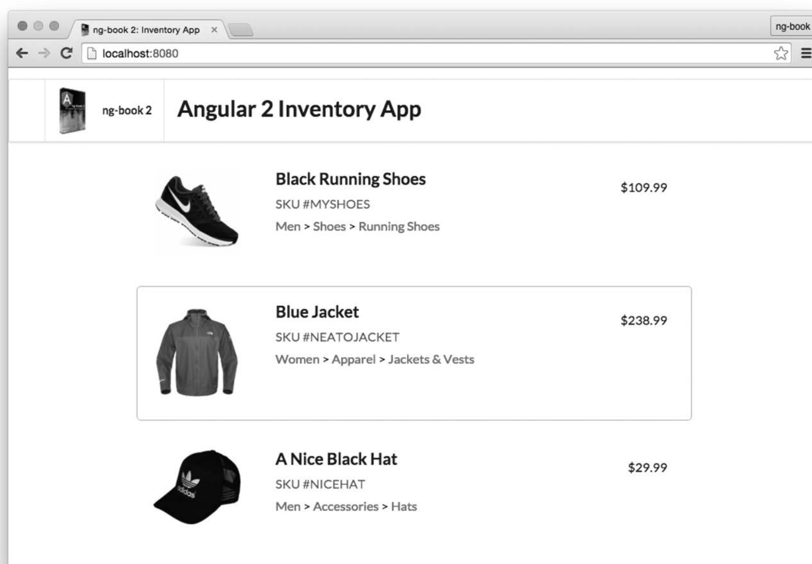


图3-7　完成的库存管理系统

3.2　产品数据模型

关于Angular，有一件事你必须清楚：它不要求使用指定的数据模型库。

Angular十分灵活，可以支持多种不同的数据模型（和数据架构）。不过这也意味着你需要决定自己的实现方式。

关于数据结构，我们会在第9章详细讲解。在本章中，我们仅使用普通的JavaScript对象作为数据模型。

code/how_angular_works/inventory_app/app.ts

```
/**
 * Provides a `Product` object
 */
class Product {
  constructor(
    public sku: string,
    public name: string,
    public imageUrl: string,
    public department: string[],
    public price: number) {
  }
}
```

如果你还不熟悉ES6/TypeScript，可能会对这段代码的语法感到陌生。

上面的代码创建了一个名叫Product的类，这个类的构造函数接收5个参数。public sku: string这行代码有两个意思：

- 这个类的实例有一个名为sku的公共属性；
- sku的类型是string。



如果你已经比较熟悉JavaScript，可以通过learnxinyminutes[1]上的教程来快速补充相关知识，比如上面代码中的public constructor简写形式。

上面代码中的Product类不依赖Angular中的任何东西，它只是一个我们会在应用中用到的数据模型。

3.3　组件

前面提到过，组件是构成Angular应用的基本组成部分。“应用”本身就是一个顶层组件，并且我们把应用划分成了细粒度的组件。

① https://learnxinyminutes.com/docs/typescript/



技巧：当开发新的Angular应用时，先画出原型图，然后拆分成组件。

因为我们经常用到组件，所以有必要对组件进行进一步研究。

每个组件都由三个部分组成：

- 组件注解
- 视图
- 控制器

要清楚这些关键概念，就要充分理解组件。我们先来分析顶层的库存管理系统应用，然后再来分析产品列表及其下级组件（如图3-8所示）。

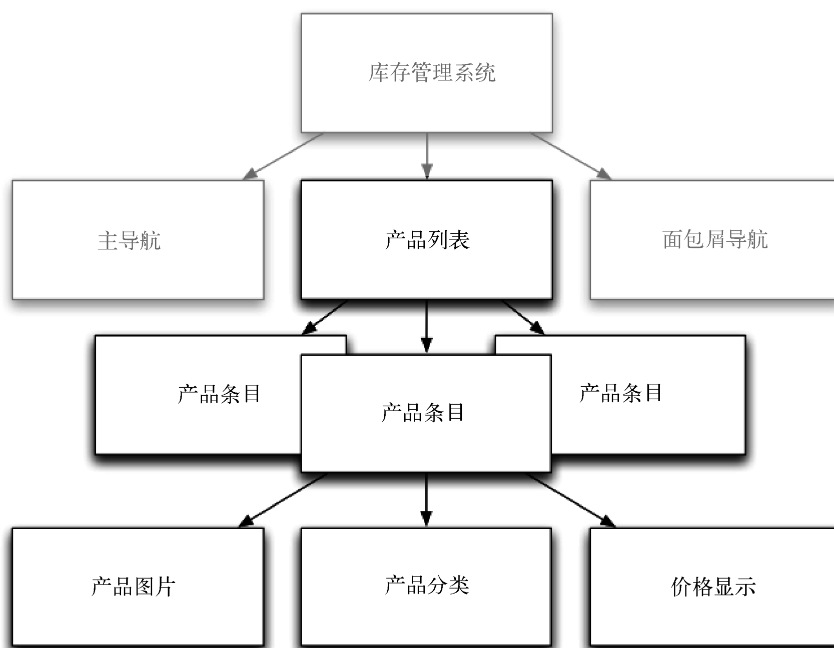


图3-8　产品列表组件

一个基本的顶层应用InventoryApp（库存管理系统）看起来是这样的：

```
@Component({
  selector: 'inventory-app',
  template: `
  <div class="inventory-app">
     (Products will go here soon)
```

```
  </div>
  `
})
class InventoryApp {
  // Inventory logic here
}

// module boot here...
```

如果你用过AngularJS，可能会觉得完全看不懂这段代码。别担心，其实两者的思路还是很相似的，让我们来一步一步地分析。

这段代码中的@Component被称作注解。它给紧随其后的类（InventoryApp）添加了一些元数据。

@Component注解明确了下面两项：

- selector（选择器）用来告诉Angular要匹配哪个HTML元素；
- template（模板）用来定义视图。

组件的控制器是由一个TypeScript类定义的，比如前面代码中的InventoryApp类。

接下来让我们对代码中的各个部分进行更详细的分析。

3.4　组件注解

@Component注解是对组件进行配置的地方。一般来说，@Component会配置你的组件如何与外界交互。

要配置一个组件，有很多种方法（我们会在第14章中进行讲解）。本章只会涉及一些基本配置。

3.4.1　组件 selector

通过selector（选择器）配置项，可以指定当HTML模板被渲染时Angular如何找到组件。这个思路与CSS、XPath中的选择器很像。我们可以用选择器来定义HTML中的哪些元素用来与组件匹配。在前面的例子中，selector: inventory-app就表示我们希望在HTML中匹配inventory-app标签。也就是说，我们定义了一个新的HTML标签，每当我们使用这个标签时，它都拥有我们定义的功能。例如，我们把下面这段代码放到HTML中：

```
<inventory-app></inventory-app>
```

Angular就会自动使用我们定义的InventoryApp组件来实现这个标签的功能。

此外，这个例子中定义的选择器还可以匹配一个以组件名为属性的普通div元素：

```
<div inventory-app></div>
```

3.4.2 组件 template

视图是一个组件中可视的部分。我们可以用@Component中的template配置项来定义组件所用的HTML模板：

```
@Component({
  selector: 'inventory-app',
  template: `
  <div class="inventory-app">
     (Products will go here soon)
  </div>
  `
})
```

可以看到，在template配置项里，我们用到了TypeScript中用反引号包裹的多行文本语法。到目前为止，我们的模板还都很简单：只有一个div和一些占位文本。



如果希望把模板放到一个单独的文件中，可以将组件的template配置项改为templateUrl配置项，把配置的内容设置为模板文件名即可。

3.4.3 添加产品

我们的应用现在还没有产品可展示，需要添加一些。

可以用如下代码创建一个Product：

```
let newProduct = new Product(
      'NICEHAT',                                  // sku
      'A Nice Black Hat',                         // name
      '/resources/images/products/black-hat.jpg', // imageUrl
      ['Men', 'Accessories', 'Hats'],             // department
      29.99);                                     // price
```

Product类的构造函数接收5个参数。新建一个Product实例要用到new关键词。



一般情况下，我们应该不会向一个函数传递超过5个参数。另一种做法是将Product类的构造函数修改为接收一个配置对象，这样就可以不必记住参数的顺序了。如果这样做，我们就可以像这样编写Product类的代码：

```
new Product({sku: "MYHAT", name: "A green hat"})
```

但就目前来说，5个参数的构造函数还可以接受。

我们希望在界面中展示这个Product。为了让产品属性在模板中可访问，我们把它们添加到组件的实例变量中。

比如，如果希望在视图中访问新产品newProduct，可以这样写：

```
class InventoryApp {
  product: Product;

  constructor() {
    let newProduct = new Product(
          'NICEHAT',
          'A Nice Black Hat',
          '/resources/images/products/black-hat.jpg',
          ['Men', 'Accessories', 'Hats'],
          29.99);

    this.product = newProduct;
  }
}
```

也可以更简洁一点：

```
class InventoryApp {
  product: Product;

  constructor() {
    this.product = new Product(
          'NICEHAT',
          'A Nice Black Hat',
          '/resources/images/products/black-hat.jpg',
          ['Men', 'Accessories', 'Hats'],
          29.99);
  }
}
```

注意，我们在这里做了三件事。

(1) 添加了一个constructor。当Angular创建这个组件的实例时，会调用这个constructor。我们可以在这里对这个组件进行初始化。

(2) 声明了一个实例变量。当我们在InventoryApp里写product : Product的时候，是在InventoryApp的实例中定义了一个名叫product的属性，用于保存Product对象。

(3) 给product属性赋值了一个Product实例。在constructor中，我们创建了一个Product的实例，并把它赋值给product实例变量。

3.4.4　用模板绑定来查看产品

由于已经给product赋了值，现在我们可以在视图中使用这个变量了。把模板修改成下面这样：

```
@Component({
  selector: 'inventory-app',
```

```
    template: `
    <div class="inventory-app">
       <h1>{{ product.name }}</h1>
       <span>{{ product.sku }}</span>
    </div>
    `
  })
```

{{...}}语法被称为模板绑定。它告诉视图，我们希望在模板的这个位置使用花括号中表达式的值。

在这个例子中，我们有两个绑定：

- {{ product.name }}
- {{ product.sku }}

product变量来自于InventoryApp组件实例中的实例变量product 。

模板绑定有个很灵活的特性：花括号中的内容是一个表达式。这意味着你可以像下面这样写代码：

- {{ count + 1 }}
- {{ myFunction(myArguments) }}

在第一个示例中，我们使用一个操作符改变了count的显示值。在第二个示例中，我们使用myFunction(myArguments)函数的返回值来作为显示内容。使用模板绑定标签是在Angular应用中展示数据的主要方式。

3.4.5　添加更多产品

我们当然不希望应用只展示一个产品；实际上，我们希望展示一个完整的产品列表。因此，把InventoryApp中的一个Product属性修改为Product数组：

```
class InventoryApp {
  products: Product[];

  constructor() {
    this.products = [];
  }
}
```

注意，我们还把product变量重命名为products，并且把类型改为了Product[]。后面的[]代表我们希望products是一个Product数组。也可以把它写成Array<Product> 。

现在InventoryApp已经可以保存多个Product了，我们在构造函数中多创建一些Product。

code/how_angular_works/inventory_app/app.ts

```
class InventoryApp {
```

```
  products: Product[];

  constructor() {
    this.products = [
      new Product(
        'MYSHOES',
        'Black Running Shoes',
        '/resources/images/products/black-shoes.jpg',
        ['Men', 'Shoes', 'Running Shoes'],
        109.99),
    new Product(
        'NEATOJACKET',
        'Blue Jacket',
        '/resources/images/products/blue-jacket.jpg',
        ['Women', 'Apparel', 'Jackets & Vests'],
        238.99),
    new Product(
        'NICEHAT',
        'A Nice Black Hat',
        '/resources/images/products/black-hat.jpg',
        ['Men', 'Accessories', 'Hats'],
        29.99)
    ];
  }
```

这段代码会在应用中创建一些产品以备后续使用。

3.4.6 选择一个产品

我们需要应用支持用户交互。比如，用户可能会希望选择一个特定的产品来查看更多信息，或者把它加入购物车，等等。

下面来给InventoryApp定义一个新方法productWasSelected，用来响应用户对产品的选择。

code/how_angular_works/inventory_app/app.ts

```
  productWasSelected(product: Product): void {
    console.log('Product clicked: ', product);
  }
```

3.4.7 用<products-list>列出产品

顶层的InventoryApp组件已经有了，现在需要创建一个新的组件用来渲染产品列表。接下来，我们会实现使用products-list选择器的ProductsList组件。在我们深入实现细节之前，先看看如何使用它。

code/how_angular_works/inventory_app/app.ts

```
@Component({
  selector: 'inventory-app',
```

```
  template: `
  <div class="inventory-app">
    <products-list
      [productList]="products"
      (onProductSelected)="productWasSelected($event)">
    </products-list>
  </div>
  `
})
class InventoryApp {
```

这里出现了一些新的语法和配置项，我们来逐一说明。

1. 输入/输出

使用products-list组件时，我们会用到Angular组件的一个核心特性：输入/输出。

```
    <products-list
      [productList]="products"                          <!-- input -->
      (onProductSelected)="productWasSelected($event)"> <!-- output -->
    </products-list>
```

方括号[]用来传递输入，圆括号()用来处理输出。

数据通过输入绑定流入你的组件，事件通过输出绑定流出你的组件。

可以将输入与输出绑定理解为对组件定义了一组公有API。

2. 方括号传递输入

在Angular中，你可以通过输入把数据传入组件。

在我们的代码中有一段：

```
    <products-list
      [productList]="products"
```

这就是在使用ProductsList组件的输入。

可能products和productList有点难以理解。这个元素属性（attribute）分为两个部分：

- [productList]（=号左边）
- "products"（=号右边）

左边的[productList]是指，我们希望在product-list组件中设置名为productList的输入。

右边的"products"是指，我们希望将输入设置为products表达式的值，即InventoryApp类中的this.products。



你可能会问："我怎么知道productList是product-list组件的一个合法输入呢？"答案是：需要阅读这个组件的相关文档。inputs（输入）和outputs（输出）是这个组件"公开API"的一部份。

你可以像弄清一个函数有哪些参数一样来弄清一个组件支持哪些输入。

3. 圆括号处理输出

在Angular中，使用输出来将数据传递出组件。

在我们的代码中有一段：

```
<products-list
  ...
  (onProductSelected)="productWasSelected($event)">
```

意思是我们要监听ProductsList组件的onProductSelected输出。

也就是说：

- (onProductSelected)，即=号左边是我们要监听的输出的名称；
- "productWasSelected"，即=号右边是当有新的输入时我们想要调用的方法；
- $event在这里是一个特殊的变量，用来表示输出的内容。

到目前为止，我们还没有讨论过如何在组件中定义输入和输出。别急，我们很快就会在定义ProductsList组件时提到这一点。

4. 完整的InventoryApp代码清单

下面是InventoryApp组件的完整代码清单。

code/how_angular_works/inventory_app/app.ts

```
@Component({
  selector: 'inventory-app',
  template: `
  <div class="inventory-app">
    <products-list
      [productList]="products"
      (onProductSelected)="productWasSelected($event)">
    </products-list>
  </div>
  `
})
class InventoryApp {
  products: Product[];

  constructor() {
    this.products = [
      new Product(
        'MYSHOES',
```

```
          'Black Running Shoes',
          '/resources/images/products/black-shoes.jpg',
          ['Men', 'Shoes', 'Running Shoes'],
          109.99),
        new Product(
          'NEATOJACKET',
          'Blue Jacket',
          '/resources/images/products/blue-jacket.jpg',
          ['Women', 'Apparel', 'Jackets & Vests'],
          238.99),
        new Product(
          'NICEHAT',
          'A Nice Black Hat',
          '/resources/images/products/black-hat.jpg',
          ['Men', 'Accessories', 'Hats'],
          29.99)
      ];
    }

    productWasSelected(product: Product): void {
      console.log('Product clicked: ', product);
    }
  }
```


3.5 产品列表组件

我们已经有了顶层应用组件，现在是时候编写用来展示产品列表的ProductsList组件了。

我们希望只允许用户选中一个Product，还希望可以知道哪个Product是用户当前选中的。ProductList组件是做这件事的绝佳场所，因为它同时“知道”所有的Product。

让我们分三步把ProductsList组件写完：

- 设置ProductsList的@Component配置项；
- 编写ProductsList的控制器类；
- 编写ProductList的视图模板。

3.5.1 设置 **ProductsList** 的@**Component** 配置项

我们来看看ProductsList的@Component配置。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductsList: A component for rendering all ProductRows and
 * storing the currently selected Product
 */
@Component({
  selector: 'products-list',
```

```
  inputs: ['productList'],
  outputs: ['onProductSelected'],
  template: `
```

在ProductsList组件代码中，最开始是我们熟悉的selector选择器配置项。这个选择器表示我们可以通过在代码中放置<products-list>标签来使用ProductsList组件。

代码中还有两处inputs和outputs配置项。

3.5.2　组件的输入

我们可以用inputs配置项来指定组件希望接收哪些参数。inputs接收一个字符串数组，用来指定输入的键（名称）。

当我们为组件指定了一个输入时，这个组件的定义类就一定要有一个实例属性来接收这个输入的值。例如，假设我们有以下代码：

```
@Component({
  selector: 'my-component',
  inputs: ['name', 'age']
})
class MyComponent {
  name: string;
  age: number;
}
```

name和age输入分别对应于MyComponent类的实例中的name和age属性。

指定组件接收一个输入参数的另一种方式是使用@Input注解。你可以先导入Input，然后把@Input()添加到属性声明上，代码如下：

```
@Component({
  selector: 'my-component'
})
class MyComponent {
  @Input() name: string;
  @Input() age: number;
}
```

如果我们要让该输入属性的内外名字不一样，可以这样写：@Input('firstname') name: String;。但是"Angular风格指南"[1]建议避免这种方式。



你可以任意选择这两种方式之一来提供输入属性，它们的效果是一样的。在本章中，我们将使用inputs: []风格，而其他章节中则使用@Input()风格。

① https://angular.io/docs/ts/latest/guide/style-guide.html

如果想使用其他模板中的MyComponent，就可以这样写：

<my-component [name]="myName" [age]="myAge"></my-component>。

注意，name属性对应name输入，也恰好与MyComponent中的name属性对应。不过这些名称并不一定要保持一致。

比如，假如我们希望标签元素的属性和组件实例中的属性使用不同的名称。也就是说，假如我们希望这个组件看起来像这样：

<my-component [shortName]="myName" [oldAge]="myAge"></my-component>

那么可以这样修改inputs配置项的字符串格式：

```
@Component({
  selector: 'my-component',
  inputs: ['name: shortName', 'age: oldAge']
})
class MyComponent {
  name: string;
  age: number;
}
```

一般而言，inputs输入字符串列表可以使用'componentProperty: exposedProperty'（'组件实例属性：标签元素属性'）的格式。

例如，我们可以像这样写一个组件：

```
@Component({
  //...
  inputs: ['name', 'age', 'enabled']
  //...
})
class MyComponent {
  name: string;
  age: number;
  enabled: boolean;
}
```

然而，如果我们希望组件实例属性enabled在组件标签中对应的标签元素属性名称为isEnabled，就可以使用上面提到的这个语法：

```
@Component({
  //...
  inputs: [
    'name: name',
    'age: age',
    'isEnabled: enabled'
  ]
  //...
})
class MyComponent {
```

```
    name: string;
    age: number;
    isEnabled: boolean;
  }
```

进一步说，由于只有一个属性需要明确指定从enabled映射到isEnabled，我们可以继续简化：

```
  @Component({
    //...
    inputs: ['name', 'age', 'isEnabled: enabled']
    //...
  })
  class MyComponent {
    name: string;
    age: number;
    isEnabled: boolean;
  }
```

在inputs输入数组中，当字符串的值是key: value（键: 值）格式的时候，含义如下：

- 键（name、age和isEnabled）表示要输入的属性在控制器看来如何（被绑定）；
- 值（name、age和enabled）表示属性在外界看来如何。

通过inputs配置项传递products

你应该还记得，在InventoryApp中，我们通过[productList]输入将products传到了products-list组件中。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @InventoryApp: the top-level component for our application
 */
@Component({
  selector: 'inventory-app',
  template: `
  <div class="inventory-app">
    <products-list
      [productList]="products"
      (onProductSelected)="productWasSelected($event)">
    </products-list>
  </div>
  `
})
class InventoryApp {
  products: Product[];

  constructor() {
    this.products = [
```

希望你现在理解了：在上面的代码中，我们是通过ProductsList组件类的一个输入参数将this.products传进去的。

3.5.3 组件的输出

如果要从组件中把数据传递出去，应该使用输出绑定。

假如我们要编写有一个按钮的组件，并且希望在这个按钮被点击的时候做点什么。

想实现这一点，只要把组件控制器中的一个方法绑定到按钮的点击输出就可以了。写法是(output)="action"。

下面是一个计数器的例子，点击按钮的时候可以对计数器进行增加或减少的操作。

```
@Component({
  selector: 'counter',
  template: `
    {{ value }}
    <button (click)="increase()">Increase</button>
    <button (click)="decrease()">Decrease</button>
  `
})
class Counter {
  value: number;

  constructor() {
    this.value = 1;
  }

  increase() {
    this.value = this.value + 1;
    return false;
  }

  decrease() {
    this.value = this.value - 1;
    return false;
  }
}
```

在这个例子中，我们希望每次点击第一个按钮的时候，调用控制器中的increase()方法。同样，每次点击第二个按钮的时候，我们希望调用decrease()方法。

圆括号属性的语法是这样的：(output)="action"。这个例子中，我们是在监听按钮的click事件。还有很多内置的事件可以监听，如mousedown、mousemove、dbl-click等。

这个例子中，事件是组件内置的。当我们编写自己的组件时，可以暴露“公开事件”（组件的outputs）来和组件外部通信。

这里要理解的关键是，在视图中，我们可以使用(output)="action"语法来监听事件。

3.5.4 触发自定义事件

上面例子中的click和mousedown等是按钮内置的事件，现在我们要来创建一个可以触发自定义事件的组件。自定义输出，我们需要做三件事：

(1) 在@Component配置中，指定outputs配置项；

(2) 在实例属性中，设置一个EventEmitter（事件触发器）；

(3) 在适当的时候，通过EventEmitter触发事件。



可能你对EventEmitter还不太熟悉，不过别担心，它并不难。EventEmitter只是一个帮你实现观察者模式[①]的对象。也就是说，它是一个管理一系列订阅者并向其发布事件的对象。就是这么简单。

来看一个使用EventEmitter的简单小例子：

```
let ee = new EventEmitter();
ee.subscribe((name: string) => console.log(`Hello ${name}`));
ee.emit("Nate");

// -> "Hello Nate"
```

当我们把一个EventEmitter赋值给一个输出的时候， Angular会自动帮我们订阅事件。我们不需要自己订阅。（当然，如果需要，你仍然可以实现自己的订阅逻辑。）

下面是一段具有outputs的组件示例代码：

```
@Component({
  selector: 'single-component',
  outputs: ['putRingOnIt'],
  template: `
    <button (click)="liked()">Like it?</button>
  `
})
class SingleComponent {
  putRingOnIt: EventEmitter<string>;

  constructor() {
    this.putRingOnIt = new EventEmitter();
  }

  liked(): void {
    this.putRingOnIt.emit("oh oh oh");
  }
}
```

① https://en.wikipedia.org/wiki/Observer_pattern

可以看到我们做了完整的三步动作：(1) 指定outputs配置项；(2) 创建一个EventEmitter并把它赋值给我们指定的输出属性putRingOnIt；(3) 当liked方法被调用时，触发这个事件。

如果希望在一个父级组件中使用这个输出，可以这样做：

```
@Component({
  selector: 'club',
  template: `
    <div>
      <single-component
        (putRingOnIt)="ringWasPlaced($event)"
        ></single-component>
    </div>
  `
})
class ClubComponent {
  ringWasPlaced(message: string) {
    console.log(`Put your hands up: ${message}`);
  }
}

// logged -> "Put your hands up: oh oh oh"
```

再来回顾一下：

- putRingOnIt是在SingleComponent的outputs配置项中定义的；
- ringWasPlaced是ClubComponent中的一个方法；
- $event包含被触发事件参数（输出的内容），在这个例子中是一个字符串。

3.5.5　编写 ProductsList 的控制器类

回到商店的例子，ProductsList控制器类需要三个实例变量：

- 一个用来保存产品列表（来自于productList 输入）；
- 一个用来输出事件（由onProductSelected触发）；
- 一个用来保存当前选中产品的引用。

下面是实现方法。

code/how_angular_works/inventory_app/app.ts

```
class ProductsList {
  /**
   * @input productList - the Product[] passed to us
   */
  productList: Product[];

  /**
   * @ouput onProductSelected - outputs the current
   *          Product whenever a new Product is selected
   */
```

```
  onProductSelected: EventEmitter<Product>;

  /**
   * @property currentProduct - local state containing
   *              the currently selected `Product`
   */
  private currentProduct: Product;

  constructor() {
    this.onProductSelected = new EventEmitter();
  }
```

可以看到，我们的productList是一个Product类型的数组，它来自于inputs。

onProductSelected是我们的输出。

currentProduct是ProductsList的一个内部属性。你可能知道它有时候被称作“组件本地状态”。它仅在组件的内部才能用到。

3.5.6　编写 ProdctsList 的视图模板

下面是products-list组件的template。

code/how_angular_works/inventory_app/app.ts

```
  template: `
  <div class="ui items">
    <product-row
      *ngFor="let myProduct of productList"
      [product]="myProduct"
      (click)='clicked(myProduct)'
      [class.selected]="isSelected(myProduct)">
    </product-row>
  </div>
  `
```

这里用到了ProductRow组件的product-row标签。我们稍后就来定义它。

我们用ngFor来迭代productsList中的每个Product。本书前面讨论过ngFor，但现在还是提醒一下。let thing of things语法是指，迭代things中的每一个元素，复制并把它赋值到变量thing中去。

因此，我们在这个例子中迭代了productList中的Products，并为每一个元素生成一个myProduct变量。



从代码风格的角度，我不会在真实应用中把这个变量命名为myProduct，而是把它叫作product甚至p。但为了把意思表达得更明确，我认为myProduct不太容易引起歧义。

有意思的是，我们甚至可以在同一个标签中使用这个myProduct变量。可以看到，接下来的三行代码里我们就是这么做的。

[product]="myProduct"是指我们要把myProduct（局部变量）传递给product-row的product输入。（我们会在下面定义ProductRow组件的时候定义这个输入。）

(click)='clicked(myProduct)'表示当元素被点击的时候我们希望做什么。click是一个内置事件，当点击宿主元素的时候就会触发。在这个例子中，当点击此元素时，就会执行ProductsList的clicked方法。

[class.selected]="isSelected(myProduct)"很有意思：Angular允许我们通过这种语法来根据不同的情况设置元素的class属性。这个语法的意思是"如果isSelected(myProduct)返回true，就给元素的CSS类增加一个selected类"。如果需要标记出当前选中的产品，这会非常好用。

你可能已经注意到了，我们还没有定义clicked和isSelected方法，那么现在就开始吧（在ProductsList中）。

1. clicked

code/how_angular_works/inventory_app/app.ts

```
clicked(product: Product): void {
  this.currentProduct = product;
  this.onProductSelected.emit(product);
}
```

该函数会做两件事：

(1) 把this.currentProduct设置为传入的Product；

(2) 将用户点击的Product从输出中传出去。

2. isSelected

code/how_angular_works/inventory_app/app.ts

```
isSelected(product: Product): boolean {
  if (!product || !this.currentProduct) {
    return false;
  }
  return product.sku === this.currentProduct.sku;
}
```

这个方法接收一个Product。如果这个product的sku与currentProduct的sku一样，就返回true；否则返回false。

3.5.7 完整的 ProductsList 组件

下面是一份完整的代码清单，我们可以看到代码的所有上下文。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductsList: A component for rendering all ProductRows and
 * storing the currently selected Product
 */
@Component({
  selector: 'products-list',
  inputs: ['productList'],
  outputs: ['onProductSelected'],
  template: `
  <div class="ui items">
    <product-row
      *ngFor="let myProduct of productList"
      [product]="myProduct"
      (click)='clicked(myProduct)'
      [class.selected]="isSelected(myProduct)">
    </product-row>
  </div>
  `
})
class ProductsList {
  /**
   * @input productList - the Product[] passed to us
   */
  productList: Product[];

  /**
   * @ouput onProductSelected - outputs the current
   *         Product whenever a new Product is selected
   */
  onProductSelected: EventEmitter<Product>;

  /**
   * @property currentProduct - local state containing
   *             the currently selected `Product`
   */
  private currentProduct: Product;

  constructor() {
    this.onProductSelected = new EventEmitter();
  }

  clicked(product: Product): void {
    this.currentProduct = product;
    this.onProductSelected.emit(product);
  }

  isSelected(product: Product): boolean {
    if (!product || !this.currentProduct) {
      return false;
    }
    return product.sku === this.currentProduct.sku;
  }

}
```

3.6　产品条目组件

ProductRow组件用于展示Product（如图3-9所示）。ProductRow有自己的模板，但也会被分成三个更小的组件：

- ProductImage，用来展示图片；
- ProductDepartment，用来展示产品分类“面包屑导航”；
- PriceDisplay，用来展示产品价格。

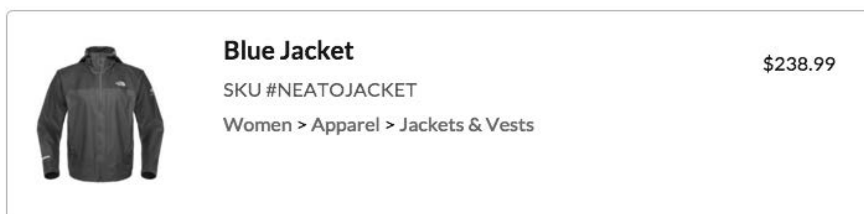


图3-9　一个被选中的ProductRow组件

可以在图3-10中看到这三个组件在ProductRow中的使用。

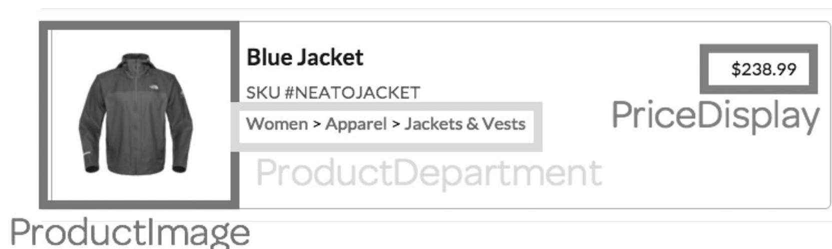


图3-10　ProductRow的子组件

下面来看看ProductRow的组件配置、定义类和模板。

3.6.1　产品条目的组件配置

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductRow: A component for the view of single Product
 */
@Component({
  selector: 'product-row',
  inputs: ['product'],
  host: {'class': 'item'},
  template: `
```

配置开头定义了product-row的selector。我们已经多次看到这个配置项了，这个定义说明

组件会匹配product-row标签。

接下来，我们定义了一个名为product的输入。这个输入就是由父级组件传入的Product。

第三个配置项host让我们可以在宿主元素上配置元素属性。在这个例子中，我们设置了Semantic UI的item样式[1]。host:{'class':'item'}的意思是，我们希望给宿主元素添加一个名为item的CSS类。



host配置项很有用，因为可以在组件内部配置宿主元素。否则必须在宿主元素的HTML标签中定义CSS等；这样，每次使用该组件时，都需要手工编写CSS类，用起来就不方便了。

我们稍后就会讨论template模板。

3.6.2　产品条目组件的定义类

ProductRow组件的定义类很简明。

code/how_angular_works/inventory_app/app.ts

```
class ProductRow {
  product: Product;
}
```

这里我们定义ProductRow会有一个实例属性product。因为我们定义了一个输入product，所以每当Angular创建这个组件的实例时，都会自动帮我们设置好product。我们不需要手动去做，也不需要constructor。

3.6.3　产品条目组件的 template

现在来看看template。

code/how_angular_works/inventory_app/app.ts

```
  template: `
  <product-image [product]="product"></product-image>
  <div class="content">
    <div class="header">{{ product.name }}</div>
    <div class="meta">
      <div class="product-sku">SKU #{{ product.sku }}</div>
    </div>
    <div class="description">
      <product-department [product]="product"></product-department>
    </div>
```

① http://semantic-ui.com/views/item.html

```
  </div>
  <price-display [price]="product.price"></price-display>
  `
```

我们的模板中没有什么新概念。

第一行使用了product-image指令，并把我们的product传递到ProductImage组件的product输入中。我们使用product-department指令时也是一样。

price-display指令的用法略有不同：我们没有直接传递product，而是传递了product.price。

剩下的模板只是带有自定义CSS样式和一些模板绑定的标准HTML元素。

3.6.4　完整的 ProductRow 代码清单

下面是ProductRow组件的全部代码。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductRow: A component for the view of single Product
 */
@Component({
  selector: 'product-row',
  inputs: ['product'],
  host: {'class': 'item'},
  template: `
  <product-image [product]="product"></product-image>
  <div class="content">
    <div class="header">{{ product.name }}</div>
    <div class="meta">
      <div class="product-sku">SKU #{{ product.sku }}</div>
    </div>
    <div class="description">
      <product-department [product]="product"></product-department>
    </div>
  </div>
  <price-display [price]="product.price"></price-display>
  `
})
class ProductRow {
  product: Product;
}
```

现在来看看我们用到的三个组件，其代码都很短。

3.7　产品图片组件

首先看看ProductImage。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductImage: A component to show a single Product's image
 */
@Component({
  selector: 'product-image',
  host: {class: 'ui small image'},
  inputs: ['product'],
  template: `
  <img class="product-image" [src]="product.imageUrl">
  `
})
class ProductImage {
  product: Product;
}
```

这里唯一需要注意的是img标签，请看看我们是怎么使用img中的[src]的。

我们本来可以这么写：

```
<!-- wrong, don't do it this way -->
<img src="{{ product.imageUrl }}">
```

为什么这样写是错的？因为如果浏览器在Angular运行起来之前就加载了这段模板，就会尝试以字符串{{ product.imageUrl }}为url来加载图片，这当然会得到一个“404 not found”错误。在Angular运行起来之前，浏览器会在页面上显示一个破损的图像。

通过[src]元素属性，我们告诉Angular我们希望使用img标签的[src]输入。一旦表达式的值解析完成，Angular就会把src元素属性替换为表达式的值。

3.8　价格展示组件

下面来看看PriceDisplay组件。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @PriceDisplay: A component to show the price of a
 * Product
 */
@Component({
  selector: 'price-display',
  inputs: ['price'],
  template: `
  <div class="price-display">\${{ price }}</div>
  `
})
class PriceDisplay {
  price: number;
}
```

这非常浅显，但要注意一点，因为在模板字符串中$是用于模板变量的特殊语法，所以在模板中出现$的写法时要进行转义。

3.9 产品分类组件

最后是ProductDepartment组件。

code/how_angular_works/inventory_app/app.ts

```
/**
 * @ProductDepartment: A component to show the breadcrumbs to a
 * Product's department
 */
@Component({
  selector: 'product-department',
  inputs: ['product'],
  template: `
  <div class="product-department">
    <span *ngFor="let name of product.department; let i=index">
      <a href="#">{{ name }}</a>
      <span>{{i < (product.department.length-1) ? '>' : ''}}</span>
    </span>
  </div>
  `
})
class ProductDepartment {
  product: Product;
}
```

这里要说明一下ProductDepartment组件中的ngFor和span标签。

我们使用了ngFor来迭代product.department中的每个分类，并赋值给name。比较新鲜的写法是第二个表达式let i=index。这是在ngFor中取得迭代序号的方法。

在span标签中，我们使用变量i来判断是否需要显示大于号>。

我们希望像这样展示分类：

```
Women > Apparel > Jackets & Vests
```

表达式{{i < (product.department.length-1) ? '>' : ''}}意味着，只要不是最后一级分类，就显示一个'>'号；如果是最后一级分类，就显示一个空字符串''。



格式test ? valueIfTrue : valueIfFalse被称作三元操作符。

3.10　创建 NgModule 并启动应用

最后要做的就是创建NgModule并启动应用。

code/how_angular_works/inventory_app/app.ts

```
@NgModule({
  declarations: [
    InventoryApp,
    ProductImage,
    ProductDepartment,
    PriceDisplay,
    ProductRow,
    ProductsList
  ],
  imports: [ BrowserModule ],
  bootstrap: [ InventoryApp ]
})
class InventoryAppModule {}
```

为了帮助我们组织代码，Angular提供了一个模块化系统。AngularJS中的所有指令本质上都是全局的，但在Angular中必须明确指出你打算在应用中使用哪些组件。

虽然使用模块系统需要更多的配置，但对于较大型的应用来说，这能避免很大的麻烦。

要使用你在Angular中创建的新组件，它们必须对于当前模块是可访问的。也就是说，如果我们要在IntentoryApp的template中通过products-list标签使用ProductsList组件的话，就要保证InventoryApp满足下面的两个条件之一：

(1) 和ProductsList组件在同一个模块中；

(2) InventoryApp所在的模块导入（imports）了ProductsList所在的模块。



记住：如果要在模板中使用，每一个组件都必须在同一个NgModule中声明。

在这个例子里，我们将InventoryApp、ProductsList和应用中的所有其他组件都放在了同一个模块中。这样写容易理解，因为它们彼此之间都是“可见”的。

注意，我们告诉NgModule要以InventoryApp来启动（bootstrap）。这就是说InventoryApp会是顶层组件。

因为我们编写的是浏览器应用，所以也把浏览器模块BrowserModule放到这个NgModule的导入列表imports里。



要了解NgModule的更多细节，请参考8.10节。

启动应用

我们现在编写的是一个没有用到AoT预编译技术（“ahead-of-time” compilation，本书后面会有详细讲解）的浏览器应用。想启动应用就要像下面这样做。

code/how_angular_works/inventory_app/app.ts

```
platformBrowserDynamic().bootstrapModule(InventoryAppModule);
```

3.11 完整的项目

现在我们已经有了让项目运行起来的所有部分！

全部完成后，应用看起来应该如图3-11所示。

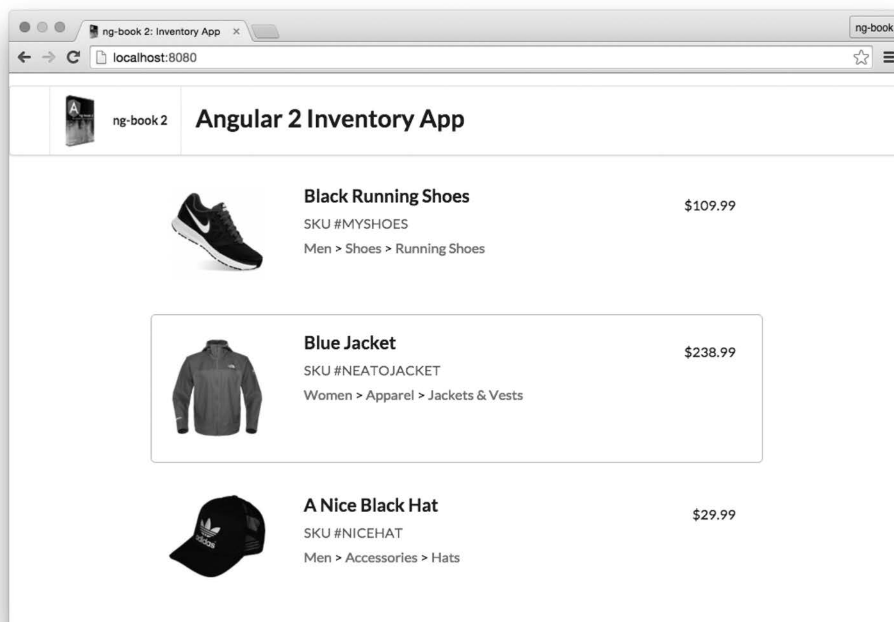


图3-11 完成后的应用



完整的代码可以在how_angular_works/inventory_app里找到，参考其中的README.md文件尝试运行。

现在你可以通过点击来选中一个特定的产品了，选中时会在外面显示一个漂亮的紫色边框。

如果你在代码中添加了新的Product，它们也会在页面中展示出来。

3.12　关于数据架构的一点说明

你可能想知道，如果开始给应用添加更多功能，该如何管理数据流呢?

例如，假设我们要加入一个购物车界面以便添加和购买商品。这该如何实现呢?

目前唯一讨论过的方案就是触发输出事件。要在点击“添加到购物车”按钮时直接把addedToCart事件冒泡上去，然后在根节点处理吗?这种做法有点怪。

数据架构是一个庞大的主题，其中存在很多不同的观点。幸运的是，Angular可以广泛适应各种数据架构，但这也意味着你需要自己选择一种。

在AngularJS中，默认选项是双向绑定。双向绑定在开发的起步阶段非常好用：控制器保存数据，表单直接修改数据，视图显示数据。

不过双向绑定的问题是，它经常导致整个应用出现级联效应。随着项目规模的扩大，我们会越来越难于追踪数据的流向。

双向绑定的另一个问题是，由于我们的数据要通过组件下发，一般情况下“数据结构树”将不得不与“DOM结构树”相对应。但在实践中，最好把这两件事分开。

处理这种情况的方法之一是创建数据服务ShoppingCartService，这是一个保存当前购物车中商品列表的单例服务。当有数据变动时，这个服务就会通知所有相关的对象。

这个主意看起来够简单了，但在实践中还有很多需要解决的问题。

Angular中推荐的方式是采用一种叫作*单向数据绑定*的方案（在其他一些现代Web开发框架中也是一样，例如React）。也就是说，你的数据只会向下流入组件。如果你需要改变数据，就要在顶层触发事件，然后向下流至底层组件。

乍看起来，单向数据绑定可能反而额外增加了一些开销，但实际上它会大幅减轻变更检测相关的复杂度，还会使你的系统行为更具可预测性。

幸运的是，数据架构管理方面只有两个主要流派：

(1) 使用基于观察者模式的架构，如RxJS；

(2) 使用基于Flux的架构。

我们稍后会讨论如何为应用实现一个可扩展的数据架构，但就目前来说，基于组件的应用已经完成了，先好好享受成功的喜悦吧!

第 4 章

内置指令

4.1 简介

Angular提供了若干内置指令。在本章中，我们将探讨每一个内置指令并通过示例教会你如何使用它们。



内置指令是已经导入过的，你的组件可以直接使用它们。因此，不用像你自己的组件一样把它们作为指令导入进来。

4.2 ngIf

如果你希望根据一个条件来决定显示或隐藏一个元素，可以使用ngIf指令。这个条件是由你传给指令的表达式的结果决定的。

如果表达式的结果返回的是一个假值，那么元素会从DOM上被移除。

下面是一些例子：

```
<div *ngIf="false"></div>          <!-- never displayed -->
<div *ngIf="a > b"></div>          <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'"></div>   <!-- displayed if str holds the string "yes" -->
<div *ngIf="myFunc()"></div>       <!-- displayed if myFunc returns a true value -->
```



如果你有AngularJS的经验，那么大概以前已经用过ngIf指令了。你可以把它当作AngularJS中ng-if的替代品。但另一方面，Angular并没有为AngularJS中的ng-show指令提供内置的替代品。那么，如果你只是想改变一个元素的CSS可见性，就应该使用ngStyle或class指令。本章稍后会介绍它们。

4.3 ngSwitch

有时候你需要根据一个给定的条件来渲染不同的元素。

遇到这种情况时，你可能会像下面这样多次使用ngIf：

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar != 'A' && myVar != 'B'">Var is something else</div>
</div>
```

如你所见，当myVar的值既不是A也不是B时，代码将变得相当繁琐，其实我们真正想表达的只是一个else而已。随着我们添加的值越来越多，ngIf条件也会变得越来越复杂。

为了说明这种增长的复杂性，假设我们想要处理一个新的值C。

为了达到目的，我们不仅要添加一个使用ngIf的新元素，而且要修改最后一种情况：

```
<div class="container">
  <div *ngIf="myVar == 'A'">Var is A</div>
  <div *ngIf="myVar == 'B'">Var is B</div>
  <div *ngIf="myVar == 'C'">Var is C</div>
  <div *ngIf="myVar != 'A' && myVar != 'B' && myVar != 'C'">Var is something else</div>
</div>
```

对于这种情况，Angular引入了ngSwitch指令。

如果你熟悉switch语句的话，应该会觉得似曾相识。

指令背后的思想也是一样的：对表达式进行一次求值，然后根据其结果来决定如何显示指令内的嵌套元素。

一旦有了结果，我们就可以：

- 使用ngSwitchCase指令描述已知结果；
- 使用ngSwitchDefault指令处理所有其他未知情况。

让我们使用这组新的指令来重写之前的例子：

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

如果想要处理新值C，只需要插入一行：

```
<div class="container" [ngSwitch]="myVar">
  <div *ngSwitchCase="'A'">Var is A</div>
  <div *ngSwitchCase="'B'">Var is B</div>
  <div *ngSwitchCase="'C'">Var is C</div>
```

```
  <div *ngSwitchDefault>Var is something else</div>
</div>
```

不需要修改默认（即备用）条件。

ngSwitchDefault元素是可选的。如果我们不用它，那么当myVar没有匹配到任何期望的值时就不会渲染任何东西。

你也可以为不同的元素声明同样的*ngSwitchCase值，这样就可以多次匹配同一个值了。例子如下：

4

code/built_in_directives/app/ts/ng_switch/ng_switch.ts

```
  template: `
    <h4 class="ui horizontal divider header">
      Current choice is {{ choice }}
    </h4>

    <div class="ui raised segment">
      <ul [ngSwitch]="choice">
        <li *ngSwitchCase="1">First choice</li>
        <li *ngSwitchCase="2">Second choice</li>
        <li *ngSwitchCase="3">Third choice</li>
        <li *ngSwitchCase="4">Fourth choice</li>
        <li *ngSwitchCase="2">Second choice, again</li>
        <li *ngSwitchDefault>Default choice</li>
      </ul>
    </div>

    <div style="margin-top: 20px;">
      <button class="ui primary button" (click)="nextChoice()">
        Next choice
      </button>
    </div>
  `
```

在上面的例子中，当choice的值是2的时候，第2个和第5个li都会被渲染。

4.4 ngStyle

使用ngStyle指令，可以通过Angular表达式给特定的DOM元素设定CSS属性。

该指令最简单的用法就是[style.<cssproperty>]="value"的形式，下面是一个例子。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
      <div [style.background-color]="'yellow'">
        Uses fixed yellow background
      </div>
```

这个代码片段就是使用ngStyle指令把CSS的background-color属性设置为字符串字面量yellow。

另一种设置固定值的方式就是使用ngStyle属性，使用键值对来设置每个属性。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```



注意，在ngStyle的说明中，我们对background-color使用了单引号，但却没有对color使用。这是为什么呢？因为ngStyle的参数是一个JavaScript对象，而color是一个合法的键，不需要引号。但是在background-color中，连字符是不允许出现在对象的键名当中的，除非它是一个字符串，因此使用了引号。
通常情况下，我尽量不会对对象的键使用引号，除非不得不用。

我们在这里同时设置了color和background-color属性。

但ngStyle指令真正的能力在于使用动态值。

在这个例子中，我们定义了两个输入框。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
<div class="ui input">
  <input type="text" name="color" value="{{color}}" #colorinput>
</div>

<div class="ui input">
  <input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>

<button class="ui primary button" (click)="apply(colorinput.value, fontinput\
.value)">
  Apply settings
</button>
```

然后使用它们的值来设置三个元素的CSS属性。

在第一个元素中，我们基于输入框的值来设定字体大小。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
<div>
  <span [ngStyle]="{color: 'red'}" [style.font-size.px]="fontSize">
    red text
  </span>
</div>
```

注意，我们在某些情况下必须指定单位。例如，把font-size设置为12不是合法的CSS，必须指定一个单位，比如12px或者1.2em。Angular提供了一个便捷语法用来指定单位：这里我们使用的格式是[style.fontSize.px]。

后缀.px表明我们设置font-size属性值以像素为单位。你完全可以把它替换为[style.font-size.em]，以相对长度为单位来表示字体大小；还可以使用[style.fontSize.%]，以百分比为单位。

另外两个元素使用#colorinput的值来设置文字颜色和背景颜色。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
<h4 class="ui horizontal divider header">
  ngStyle with object property from variable
</h4>

<div>
  <span [ngStyle]="{color: color}">
    {{ color }} text
  </span>
</div>

<h4 class="ui horizontal divider header">
  style from variable
</h4>

<div [style.background-color]="color"
     style="color: white;">
  {{ color }} background
</div>
```

这样，当我们点击Apply settings按钮时，就会调用方法来设置新的值。

code/built_in_directives/app/ts/ng_style/ng_style.ts

```
apply(color: string, fontSize: number) {
  this.color = color;
  this.fontSize = fontSize;
}
```

与此同时，文本颜色和字体大小都通过NgStyle指令作用在元素上了。

4.5 ngClass

ngClass指令在HTML模板中用ngClass属性来表示，让你能动态设置和改变一个给定DOM元素的CSS类。



如果你用过AngularJS，会发现ngClass指令和过去在AngularJS中的ngClass所做的事是非常相似的。

使用这个指令的第一种方式是传入一个对象字面量。该对象希望以类名作为键，而值应该是一个用来表明是否应该应用该类的真/假值。

假设我们有一个叫作bordered的CSS类，用来给元素添加一个黑色虚线边框。

code/built_in_directives/app/css/styles.scss

```
.bordered {
  border: 1px dashed black;
  background-color: #eee;
}
```

我们来添加两个div元素：一个一直都有bordered类（因此一直有边框），而另一个永远都不会有。

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
```

如预期一样，两个div应该是如图4-1这样渲染的。



图4-1　ngClass指令的简单用法

当然，使用ngClass指令来动态分配类会有用得多。

为了动态使用它，我们添加了一个变量作为对象的值：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div [ngClass]="{bordered: isBordered}">
 Using object literal. Border {{ isBordered ? "ON" : "OFF" }}
</div>
```

或者在组件中定义该对象：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
export class NgClassSampleApp {
  isBordered: boolean;
  classesObj: Object;
  classList: string[];
```

并直接使用它：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div [ngClass]="classesObj">
  Using object var. Border {{ classesObj.bordered ? "ON" : "OFF" }}
</div>
```



再次强调，当你使用像bordered-box这种包含连字符的类名时需要小心。JavaScript对象不允许字面量的键出现连字符。如果确实需要，那就必须像这样用字符串作为键：

```
<div [ngClass]="{'bordered-box': false}">...</div>
```

我们也可以使用一个类名列表来指定哪些类名会被添加到元素上。为此，我们可以传入一个数组型字面量：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div class="base" [ngClass]="['blue', 'round']">
  This will always have a blue background and
  round corners
</div>
```

或者在组件中声明一个数组对象：

```
  this.classList = ['blue', 'round'];
```

并把它传进来：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div class="base" [ngClass]="classList">
  This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
  and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
</div>
```

在上个例子中，[ngClass]分配的类名和通过HTML的class属性分配的已存在类名都是生效的。

最后添加到元素的类总会是HTML属性class中的类和[ngClass]指令求值结果得到的类的集合。

在这个例子中：

code/built_in_directives/app/ts/ng_class/ng_class.ts

```
<div class="base" [ngClass]="['blue', 'round']">
  This will always have a blue background and
  round corners
</div>
```

元素有全部的三个类：HTML的class属性提供的base，以及通过[ngClass]分配的blue和round（如图4-2所示）。



图4-2 来自属性和指令的CSS类

4.6 ngFor

这个指令的任务是重复一个给定的DOM元素（或一组DOM元素），每次重复都会从数组中取一个不同的值。



这个指令是AngularJS中ng-repeat的继任者。

它的语法是`*ngFor="let item of items"`。

- ❑ `let item`语法指定一个用来接收`items`数组中每个元素的（模板）变量。
- ❑ `items`是来自组件控制器的一组项的集合。

要阐明这一点，我们来看一下代码示例。我们在组件控制器中声明了一个城市的数组：

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

然后在模板中有如下的HTML片段。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
<h4 class="ui horizontal divider header">
  Simple list of strings
</h4>

<div class="ui list" *ngFor="let c of cities">
  <div class="item">{{ c }}</div>
</div>
```

它会如你期望的那样在`div`中渲染每一个城市，如图4-3所示。

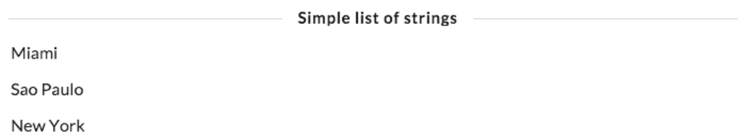


图4-3 使用ngFor指令的结果

我们还可以这样迭代一个对象数组。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
    this.people = [
      { name: 'Anderson', age: 35, city: 'Sao Paulo' },
      { name: 'John', age: 12, city: 'Miami' },
      { name: 'Peter', age: 22, city: 'New York' }
    ];
```

然后根据每一行数据渲染出一个表格。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
  <h4 class="ui horizontal divider header">
    List of objects
  </h4>

  <table class="ui celled table">
    <thead>
      <tr>
        <th>Name</th>
        <th>Age</th>
        <th>City</th>
      </tr>
    </thead>
    <tr *ngFor="let p of people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
      <td>{{ p.city }}</td>
    </tr>
  </table>
```

结果如图4-4所示。

List of objects

Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

图4-4　渲染对象数组

我们还可以使用嵌套数组。如果想根据城市进行分组，可以定义一个新对象数组。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
    this.peopleByCity = [
      {
```

```
        city: 'Miami',
        people: [
          { name: 'John', age: 12 },
          { name: 'Angel', age: 22 }
        ]
      },
      {
        city: 'Sao Paulo',
        people: [
          { name: 'Anderson', age: 35 },
          { name: 'Felipe', age: 36 }
        ]
      }
    ];
  };
```

然后可以使用ngFor为每个城市渲染一个h2标签。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>
```

并且使用一个嵌套指令对这个城市中的人们进行迭代。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tr *ngFor="let p of item.people">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
  </tr>
</table>
```

下面是模板代码的最终结果。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
<h4 class="ui horizontal divider header">
  Nested data
</h4>

<div *ngFor="let item of peopleByCity">
  <h2 class="ui header">{{ item.city }}</h2>

  <table class="ui celled table">
    <thead>
      <tr>
        <th>Name</th>
```

```
        <th>Age</th>
      </tr>
    </thead>
    <tr *ngFor="let p of item.people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
    </tr>
  </table>
</div>
```


它会为每个城市渲染一个表格，如图4-5所示。

Nested data

Miami

Name	Age
John	12
Angel	22

Sao Paulo

Name	Age
Anderson	35
Felipe	36

图4-5　渲染嵌套数组

获取索引

在迭代数组时，我们可能也要获取每一项的索引。

我们可以在ngFor指令的值中插入语法`let idx = index`并用分号分隔开，这样就可以获取索引了。这时候，Angular会把当前的索引分配给我们提供的变量（在这里是变量`idx`）。



注意，和JavaScript一样，索引都是从0开始的。因此第一个元素的索引是0，第二个是1，以此类推。

对我们的第一个例子稍加改动，添加代码段`let num = index`。

code/built_in_directives/app/ts/ng_for/ng_for.ts

```
<div class="ui list" *ngFor="let c of cities; let num = index">
  <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

它会在城市的名称前面添加序号，如图4-6所示。

List with index

1 - Miami

2 - Sao Paulo

3 - New York

图4-6　使用索引

4.7　ngNonBindable

当我们想告诉Angular不要编译或者绑定页面中的某个特殊部分时，要使用ngNodBindable指令。

假设我们想在模板中渲染纯文本{{ content }}。通常情况下，这段文本会被绑定到变量content的值，因为我们使用了{{ }}模板语法。

那该如何渲染出纯文本{{ content }}呢？可以使用ngNonBindable指令。

假设我们想要用一个div来渲染变量content的内容，紧接着输出文本<- this is what {{ content }} rendered来指向变量实际的值。

为了做到这一点，要使用下面的模板。

code/built_in_directives/app/ts/ng_non_bindable/ng_non_bindable.ts

```
template: `
<div class='ngNonBindableDemo'>
  <span class="bordered">{{ content }}</span>
  <span class="pre" ngNonBindable>
    &larr; This is what {{ content }} rendered
  </span>
</div>
`
```

有了ngNonBindable属性，Angular不会编译第二个span里的内容，而是原封不动地将其显示出来（如图4-7所示）。

Some text ← This is what {{ content }} rendered

图4-7　使用ngNonBindable的结果

4.8　总结

Angular的核心指令数量很少，但我们却能通过组合这些简单的指令来创建五花八门的应用。

第 5 章

Angular中的表单

5

5.1 表单——既重要，又复杂

在Web应用中，表单或许是最重要的部分。虽然我们常从点击链接或移动鼠标中得到事件通知，但大多数“富数据”都是通过表单从用户那里获得的。

从表面上看，表单似乎很简单：创建一个input标签，用户填入数据，然后再点击提交。这有什么难的？

但事实证明，表单最终可能是非常复杂的。原因如下：

- 表单输入意味着需要在页面和服务器端同时修改这份数据；
- 修改的内容通常要在页面的其他地方反映出来；
- 用户的输入可能存在很多问题，所以需要验证输入的内容；
- 用户界面需要清晰地显示出可能出现的预期结果和错误信息；
- 字段之间的依赖可能存在复杂的业务逻辑；
- 我们希望不依赖DOM选择器就能轻松测试表单。

值得庆幸的是，Angular已经给出了上述所有问题的解决方案。

- 表单控件（FormControl）封装了表单中的输入，并提供了一些可供操纵的对象。
- 验证器（validator）让我们能以自己喜欢的任何方式验证表单输入。
- 观察者（observer）让我们能够监听表单的变化，并作出相应的回应。

在本章中，我们将一步一步构建表单应用。先构建一些简单的表单，然后构建逻辑更复杂的表单。

5.2 FormControl 和 FormGroup

FormControl和FormGroup是Angular中两个最基础的表单对象。

5.2.1 FormControl

FormControl代表单一的输入字段，它是Angular表单中的最小单元。

FormControl封装了这些字段的值和状态，比如是否有效、是否脏（被修改过）或是否有错误等。

比如，下列代码演示了如何在TypeScript中使用FormControl：

```
// create a new FormControl with the value "Nate"
let nameControl = new FormControl("Nate");

let name = nameControl.value; // -> Nate

// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
// etc.
```

为了构建表单，我们会创建几组FormControl对象，然后为它们附加元数据和逻辑。

在Angular中，我们经常将一个类（本例中为FormControl）以属性形式（本例中为formControl）附加在DOM上。比如下面这个表单：

```
<!-- part of some bigger form -->
<input type="text" [formControl]="name" />
```

这会在此form的上下文中创建一个新的FormControl对象。稍后我们会进一步讨论它的工作原理。

5.2.2 FormGroup

大多数表单都拥有不止一个字段，因此我们需要某种方式来管理多个FormControl。假设我们要检查表单的有效性。如果要遍历这个FormControl数组并检查每一个FormControl是否有效，必然相当繁琐；而FormGroup则可以为一组FormControl提供总包接口（wrapper interface），来解决这种问题。

下面是FormGroup的创建方式：

```
let personInfo = new FormGroup({
    firstName: new FormControl("Nate"),
    lastName: new FormControl("Murray"),
    zip: new FormControl("90210")
})
```

FormGroup和FormControl都继承自同一个祖先AbstractControl[①]。这意味检查personInfo

① https://github.com/angular/angular/blob/master/modules/angular2/src/common/forms/model.ts

的状态或值就像检查单个FormControl那么容易：

```
personInfo.value; // -> {
//   firstName: "Nate",
//   lastName: "Murray",
//   zip: "90210"
// }

// now we can query this control group for certain values, which have sensible
// values depending on the children FormControl's values:
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty  // -> false
personInfo.valid  // -> true
// etc.
```


注意，当我们试图从FormGroup中获取value时，会收到一个“键值对”结构的对象。它能让我们从表单中一次性获取全部的值而无需逐一遍历FormControl，使用起来相当顺手。

5.3　我们的第一个表单

创建表单的方式很多，而且好几种重要的方式我们还没有讨论到。先来看一个完整的例子，稍后再一一解释。



本节的所有示例代码都可以在forms/目录下找到。

我们要构建的第一个表单，效果如图5-1所示。

Demo Form: Sku

SKU

SKU

Submit

图5-1　带SKU的表单演示：简易版

假设我们要构建一个电子商务网站来展示并销售一些产品。在此应用中需要存储产品的SKU，因此先来创建一个只有SKU输入框的简易表单。



SKU是库存单位（stockkeeping unit）的缩写。它是用来跟踪产品库存的唯一编号。当我们提到SKU时，指的是人类可读的产品编码。

这个表单超级简单：只有一个sku（带label）输入框和一个提交按钮。

我们先把表单变为组件。你应该还记得，定义组件需要包含以下三个部分：

- ❑ 配置@Component()注解；
- ❑ 创建模板；
- ❑ 在组件定义类中实现自定义功能。

下面来依次实现它们。

5.3.1　加载 FormsModule

为了使用这个新的表单库，先要确保我们的NgModule中导入了这个表单库。

Angular中有两种使用表单的方式，我们在本章中都会展开讨论：使用FormsModule以及使用ReactiveFormsModule。既然都要用到，那么这个模块就同时导入它们。因此需要在引用启动程序app.ts中这样写：

```
import {
  FormsModule,
  ReactiveFormsModule
} from '@angular/forms';

// farther down...

@NgModule({
  declarations: [
    FormsDemoApp,
    DemoFormSku,
    // ... our declarations here
  ],
  imports: [
    BrowserModule,
    FormsModule,         // <-- add this
    ReactiveFormsModule  // <-- and this
  ],
  bootstrap: [ FormsDemoApp ]
})
class FormsDemoAppModule {}
```

这确保了我们能在视图中使用Angular表单指令。先简要介绍一下，FormsModule为我们提供了一些模板驱动的指令，例如：

- ❑ ngModel
- ❑ NgForm

ReactiveFormsModule则提供了下列指令：

- ❑ formControl
- ❑ ngFormGroup

此外，还有很多指令。我们还没有讨论过如何使用这些指令以及它们是做什么的，但很快就要讲到了。现在只需要知道把FormsModule和ReactiveFormsModule导入到我们的NgModule中就行了。这表示我们能在视图中使用上述所有指令，并能在组件中注入相应的服务。

5.3.2　简易 SKU 表单：@Component 注解

现在我们就可以开始创建组件了。

code/forms/app/forms/demo_form_sku.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'demo-form-sku',
```

这里定义了一个demo-form-sku的选择器（selector）。还记得吧？selector会告诉Angular，组件将绑定到哪些元素上。这里我们可以通过demo-form-sku标签来使用这个组件：

```
<demo-form-sku></demo-form-sku>
```

5.3.3　简易 SKU 表单：template

我们来看看template。

code/forms/app/ts/forms/demo_form_sku.ts

```
  template: `
  <div class="ui raised segment">
    <h2 class="ui header">Demo Form: Sku</h2>
    <form #f="ngForm"
          (ngSubmit)="onSubmit(f.value)"
          class="ui form">

      <div class="field">
        <label for="skuInput">SKU</label>
        <input type="text"
               id="skuInput"
               placeholder="SKU"
               name="sku" ngModel>
      </div>

      <button type="submit" class="ui button">Submit</button>
    </form>
  </div>
  `
```

1. form和NgForm

现在事情开始变得有趣了：我们导入了FormsModule，因此可以在视图中使用NgForm了。记住，当这些指令在视图中可用时，它就会被附加到任何能匹配其selector的节点上。

NgForm做了一件便利但隐晦的工作：它的选择器包含form 标签（而不用显式添加ngForm属性）。这意味着当我们导入FormsModule时候，NgForm就会被自动附加到视图中所有的<form>标签上。这确实非常有用，但由于它发生在幕后，也许会让很多人感到困惑。

NgForm给我们提供了两个重要的功能：

(1) 一个名叫ngForm的FormGroup对象；

(2) 一个输出事件(ngSubmit)。

你可以看到我们在视图的<form>标签中同时用到了它们两个。

code/forms/app/ts/forms/demo_form_sku.ts

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
```

首先，我们使用了#f="ngForm"。#v=thing语法的意思是，我们希望在当前视图中创建一个局部变量。

这里我们为视图中的ngForm创建了一个别名，并绑定到变量#f。这个ngForm来自哪里呢？它是由NgForm指令导出的。

ngForm是什么类型的对象呢？它是FormGroup类型的。这意味着我们可以在视图中把变量f当作FormGroup使用，而这也正是我们在输出事件(ngSubmit)中的使用方法。



细心的读者可能会注意到，上面提到NgForm会自动附加到<form>标签上（因为NgForm指令的选择器中默认包含了form），这意味着我们不必添加ngForm属性就能使用NgForm指令。但是这里我们也将ngForm添加到了属性的值上。这是笔误吗？

不，这不是笔误。如果ngForm是属性的键，那就是在告诉Angular：我们要根据这个属性使用NgForm指令。但在这里，我们要对一个**引用**赋值，而把ngForm用作**属性值**。这表示把ngForm这个表达式的执行结果赋值给局部模板变量f。

既然ngForm在这个节点上，你应该可以推断出我们正在导出的这个f变量是FormGroup类型的，接下来就可以在视图中的任何地方引用它了。

我们在表单中绑定ngSubmit事件的语法是：(ngSubmit)="onSubmit(f.value)"。

- (ngSubmit)：来自NgForm指令。
- onSubmit()：将会在我们的组件类中进行定义（稍后）。
- f.value：f就是我们前面提到的FormGroup，而.value会以键值对的形式返回FormGroup中所有控件的值。

总结起来，这行代码的意思是：“当我提交表单时，将会以该表单的值作为参数，调用组件实例上的onSubmit方法。”

2. input和NgModel

在讨论NgModel之前，关于input标签还有几点需要说明。

code/forms/app/ts/forms/demo_form_sku.ts

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
      class="ui form">

  <div class="field">
    <label for="skuInput">SKU</label>
    <input type="text"
           id="skuInput"
           placeholder="SKU"
           name="sku" ngModel>
  </div>
```

- class="ui form"和class="field"是两个可选的类。它们来自CSS框架Semantic UI[①]。它们并不属于Angular的范畴，在这里加上它们只是为了让本例子好看一些。
- label标签的for属性和input标签的id属性是一致的，这依据的是W3C标准[②]。
- 我们设置SKU控件的placeholder属性，将其作为input值为空时给用户的提示。

NgModel指令指定的selector是ngModel。这意味着我们可以通过添加这个属性把它附加到input标签上：ngModel="whatever"。在这里我们指定了一个不带属性值的ngModel。

有两种不同的方法能在模板中指定ngModel，这里是第一种。当使用不带属性值的ngModel时，我们是要指定：

(1) 单向数据绑定；

(2) 希望在表单中创建一个名为sku的FormControl（这个sku来自input标签上的name属性）。

NgModel会创建一个新的FormControl对象，把它自动添加到父FormGroup上（这里也就是form表单对象），并把这个FormControl对象绑定到一个DOM上。也就是说，它会在视图中的input标签和FormControl对象之间建立关联。这种关联是通过name属性建立的，在本例中是"sku"。



NgModel与ngModel有什么不同呢？通常，我们使用Pascal命名法（如NgModel）时，指的是类和供代码中引用的对象。首字母小写的驼峰命名法（如ngModel）来自指令的选择器selector，并且只会被用在DOM/模板中。
需要指出的是，NgModel和FormControl并不是同一个。NgModel是用在视图中的指令， 而FormControl则用来表示表单中的数据和验证规则。

① http://semantic-ui.com/

② http://www.w3.org/TR/WCAG20-TECHS/H44.html



有时，我们希望用ngModel来实现AngularJS那样的**双向绑定**。在本章的最后，我们会看到如何进行实现。

5.3.4　简易 SKU 表单：组件定义类

现在来看看组件类的定义。

code/forms/app/ts/forms/demo_form_sku.ts

```
export class DemoFormSku {
  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }
}
```

在这里，我们的类定义了一个名为onSubmit的方法，该方法会在表单提交时调用。目前我们只用console.log打印出传进去的值。

5.3.5　试试看

全部代码如下所示。

code/forms/app/ts/forms/demo_form_sku.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'demo-form-sku',

  template: `
  <div class="ui raised segment">
    <h2 class="ui header">Demo Form: Sku</h2>
    <form #f="ngForm"
          (ngSubmit)="onSubmit(f.value)"
          class="ui form">

      <div class="field">
        <label for="skuInput">SKU</label>
        <input type="text"
               id="skuInput"
               placeholder="SKU"
               name="sku" ngModel>
      </div>

      <button type="submit" class="ui button">Submit</button>
    </form>
  </div>
  `
})
```

```
export class DemoFormSku {
  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }
}
```

如果打开浏览器运行代码，结果如图5-2所示。

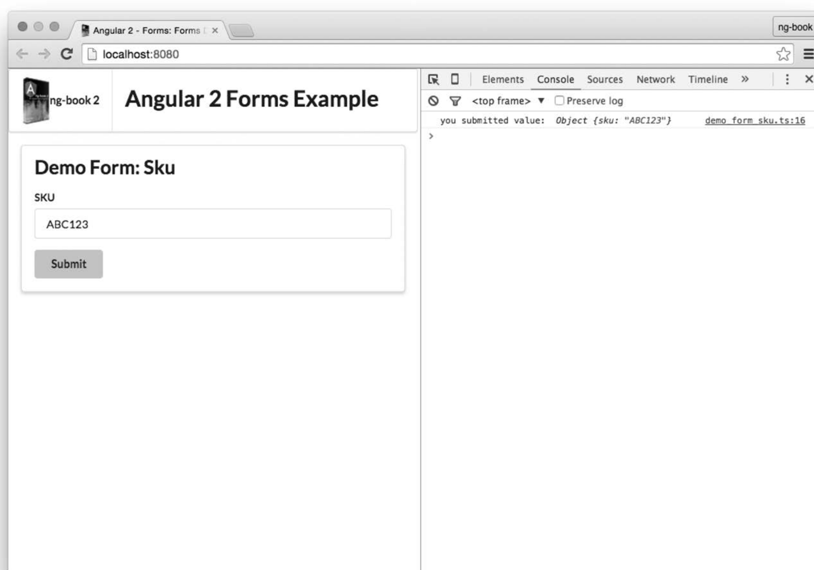


图5-2　带SKU的表单演示：简易版，已提交

5.4　使用 FormBuilder

使用ngForm和ngControl隐式构建FormControl和FormGroup确实很方便，但无法为我们提供更多定制化选项。使用FormBuilder构建表单则是一种更为灵活和通用的方式。

FormBuilder是一个名副其实的表单构建助手。你应该还记得，表单是由FormControl和FormGroup构成的，而FormBuilder则可以帮助我们创建它们（你可以把它看作一个“工厂”对象）。

让我们在先前的例子中添加一个FormBuilder，看看：

- ❑ 如何在组件定义类中使用FormGroup；
- ❑ 如何在视图表单中使用自定义的FormGroup。

5.5 响应式表单 FormBuilder

我们将使用formGroup和formControl指令来构建这个组件，这意味着我们需要导入相应的类。导入的代码如下所示。

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup
} from '@angular/forms';

@Component({
  selector: 'demo-form-sku-builder',
```

5.5.1 使用 FormBuilder

通过在组件类上声明带参数的constructor，我们注入了一个FormBuilder。

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
export class DemoFormSkuBuilder {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```



注入意味着什么？我们还未曾深入讨论过依赖注入（dependency injection，DI）以及DI是如何关联到继承树的，因此你可能看不太懂最后这句话。我们在第8章中讨论了很多关于依赖注入的知识，如果你希望深入学习，请移步那里。
大体来说，依赖注入就是用来告诉Angular，为了让组件正常运行需要给它哪些依赖。

在这期间，Angular将会注入一个从FormBuilder类创建的对象实例，并把它赋值给fb变量（来自构造函数）。

我们将会使用FormBuilder中的两个主要函数：

- ❑ control，用于创建一个新的FormControl；
- ❑ group，用于创建一个新的FormGroup。

注意，我们在类中创建了一个名叫myForm的实例变量。（简单起见，确实也可以把它称作form，但这里是为了区分FormGroup和之前的form表单。）

myForm是FormGroup类型。我们通过调用fb.group()来创建FormGroup。.group方法的参数是代表组内各个FormControl的键值对。

在这里，我们设置了一个名为sku的控件，其值为["ABC123"]——意思是控件的默认值为"ABC123"。（你可能注意到了这里用的是数组。这是因为我们稍后还会添加更多配置项。）

现在我们就能在视图中使用myForm了。（也就是说，我们需要将它绑定到表单元素上。）

5.5.2 在视图中使用 myForm

我们希望修改<form>标签，让它使用myForm变量。回忆一下，在上一节中我们提到过，当导入FormsModule时，ngForm就会自动起作用。还提到过ngForm会自动创建它自己的FormGroup。但在这里我们不希望使用外部的FormGroup，而是使用FormBuilder创建的这个myForm实例变量。那该怎么做呢？

Angular提供了另一个指令，能让我们使用现有的FormGroup。它叫作formGroup，可以这样使用。

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
<h2 class="ui header">Demo Form: Sku with Builder</h2>
<form [formGroup]="myForm"
```

这里我们告诉Angular，想用myForm作为这个表单的FormGroup。



我们说过，当使用FormsModule时，NgForm会自动应用于<form>元素上。但其实有一个例外：NgForm不会应用到带formGroup属性的<form>节点上。
你也许不明白原因，这是因为NgForm的selector是：

```
form:not([ngNoForm]):not([formGroup]),ngForm,[ngForm]
```

这意味着你还**可以**使用ngNoForm属性产生一个不带NgForm的<form>表单。

我们还需要把onSubmit中的f替换为myForm，因为现在的myForm变量中保存着表单的配置和值。

想让程序运行起来，还要做最后一件事：将我们的FormControl绑定到input标签上。记住，ngControl会创建一个新的FormControl对象，并附加到父FormGroup中。但在这个例子中，我们

已经用FormBuilder创建了自己的FormControl。

要将现有的FormControl绑定到input上，可以用formControl。

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
          <label for="skuInput">SKU</label>
          <input type="text"
                 id="skuInput"
                 placeholder="SKU"
                 [formControl]="myForm.controls['sku']">
```

在这里，我们将input标签上的formControl指令指向了myForm.controls上现有的FormControl控件sku。

5.5.3　试试看

将上面的所有代码整合在一起。

code/forms/app/ts/forms/demo_form_sku_with_builder.ts

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup
} from '@angular/forms';

@Component({
  selector: 'demo-form-sku-builder',
  template: `
  <div class="ui raised segment">
    <h2 class="ui header">Demo Form: Sku with Builder</h2>
    <form [formGroup]="myForm"
          (ngSubmit)="onSubmit(myForm.value)"
          class="ui form">

      <div class="field">
        <label for="skuInput">SKU</label>
        <input type="text"
               id="skuInput"
               placeholder="SKU"
               [formControl]="myForm.controls['sku']">
      </div>

    <button type="submit" class="ui button">Submit</button>
    </form>
  </div>
  `
})
export class DemoFormSkuBuilder {
  myForm: FormGroup;
```

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

你需要记住以下两点。

如果想隐式创建新的FormGroup和FormControl，使用：

- ngForm
- ngModel

如果要绑定一个现有的FormGroup和FormControl，使用：

- formGroup
- formControl

5.6 添加验证

用户输入的数据格式并不总是正确的。如果有人输入了错误的数据格式，我们希望给他反馈，并阻止他提交表单。因此，我们要用到验证器。

验证器由Validators模块提供。Validators.required是最简单的验证，表明指定的字段是必填项，否则就认为这个FormControl是无效的。

想使用验证器，我们得做两件事：

(1) 为FormControl对象指定一个验证器；

(2) 在视图中检查验证器的状态，并据此采取行动。

要为FormControl对象分配一个验证器，我们可以直接把它作为第二个参数传给FormControl的构造函数。

```
let control = new FormControl('sku', Validators.required);
```

也可以像这个例子中一样通过如下语法使用FormBuilder。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });
```

```
    this.sku = this.myForm.controls['sku'];
  }
```

现在要在视图中使用验证了。在视图中访问验证的值有以下两种方法。

(1) 我们可以显式地把sku这个FormControl赋值给类的实例变量。这有点啰嗦，但便于我们在视图中访问这个FormControl 。

(2) 我们也可以在myForm中查找sku这个FormControl。这样能简化组件类中的工作，但在视图中会稍微麻烦些。

为了说明两者之间的差异，我们来看两个例子。

5.6.1　显式地把 sku 设置为实例变量

图5-3展示了这个带验证功能的表单应该是什么样子的。

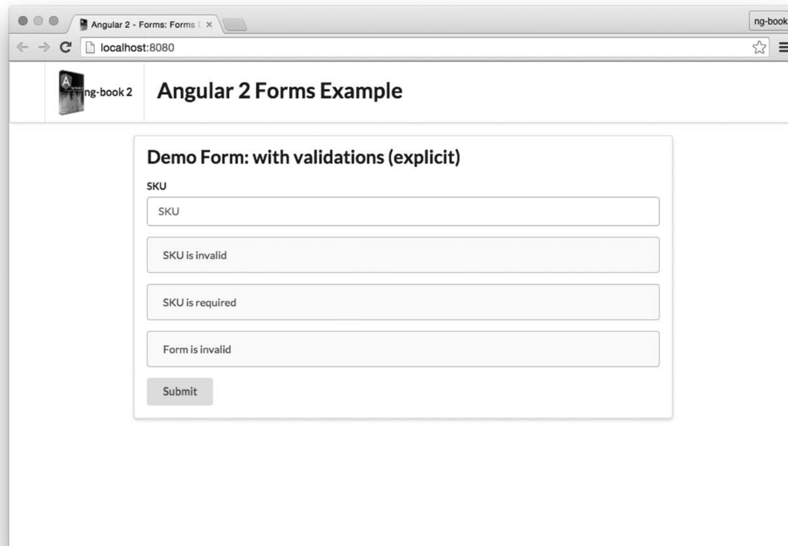


图5-3　带验证器的演示表单

在视图中，处理单个FormControls的最灵活的方式是将每个FormControl都定义在组件类上。把sku定义在类上的代码如下所示。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
export class DemoFormWithValidationsExplicit {
  myForm: FormGroup;
```

```
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

注意：

(1) 我们在类的顶部设置sku: AbstractControl；

(2) 我们把用FormBuilder创建的myForm赋值给this.sku变量。

非常好，这意味着我们可以在组件视图中到处引用sku了。不过这样做有一个缺点：我们不得不为表单中的每个字段定义一个实例变量。对大型表单而言，这会显得相当啰嗦。

现在我们的sku可以得到验证了。我们要以四种不同的方式把它用在视图中：

(1) 检查整个表单的有效性并显示一条错误信息；

(2) 检查单个字段的有效性并显示一条错误信息；

(3) 检查单个字段的有效性，当字段无效时将字段显示为红色；

(4) 检查单个字段在特定规则下的有效性并显示一条错误信息。

1. 表单信息

我们可以通过myForm.valid来检查整个表单的有效性。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
<div *ngIf="!sku.valid"
  class="ui error message">SKU is invalid</div>
```

记住，myForm是一个FormGroup；只有当里面所有的FormControl都有效时，这个FormGroup才有效。

2. 字段信息

当字段的FormControl无效时，我们也可以为该字段显示一条错误信息。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
<div *ngIf="!sku.valid"
  class="ui error message">SKU is invalid</div>
```

```
        <div *ngIf="sku.hasError('required')"
          class="ui error message">SKU is required</div>
```

3. 字段着色

这里用的是Semantic UI CSS框架的CSS类.error。当给<div class= "field">节点加上CSS类error时，这个输入框就会带有红色的边框。

我们可以使用这种“属性语法”来有条件地设置这个CSS类。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
    <div class="field"
      [class.error]="!sku.valid && sku.touched">
```

注意，这里我们为.error类设置了两个条件：检查!sku.valid和sku.touched。这是因为我们希望只有当用户修改过表单后（touched）才显示错误状态。

试着在input标签中输入一些数据，然后删除这个字段的内容。

4. 特定验证

可能有很多原因导致一个表单字段无效。对于失败的验证，我们通常希望根据不同的原因显示不同的消息。

我们可以用hasError方法来检查特定的验证失败。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
        <div *ngIf="sku.hasError('required')"
          class="ui error message">SKU is required</div>
```

注意，FormControl和FormGroup都定义了hasError方法。这意味着我们可以给它传入第二个参数path来在FormGroup中查询特定的字段。比如可以这样写：

```
        <div *ngIf="myForm.hasError('required', 'sku')"
          class="error">SKU is required</div>
```

5. 整合

下面是我们把FormControl用作实例变量来实现验证功能的完整代码。

code/forms/app/ts/forms/demo_form_with_validations_explicit.ts

```
/* tslint:disable:no-string-literal */
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators,
  AbstractControl
} from '@angular/forms';

@Component({
```

```
  selector: 'demo-form-with-validations-explicit',
  template: `
  <div class="ui raised segment">
    <h2 class="ui header">Demo Form: with validations (explicit)</h2>
    <form [formGroup]="myForm"
          (ngSubmit)="onSubmit(myForm.value)"
          class="ui form">

      <div class="field"
          [class.error]="!sku.valid && sku.touched">
        <label for="skuInput">SKU</label>
        <input type="text"
               id="skuInput"
               placeholder="SKU"
               [formControl]="sku">
          <div *ngIf="!sku.valid"
            class="ui error message">SKU is invalid</div>
          <div *ngIf="sku.hasError('required')"
            class="ui error message">SKU is required</div>
      </div>

      <div *ngIf="!myForm.valid"
        class="ui error message">Form is invalid</div>

        <button type="submit" class="ui button">Submit</button>
    </form>
  </div>
  `
})
export class DemoFormWithValidationsExplicit {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

6. 移除sku实例变量

在上面的例子中，我们将sku: AbstractControl设置为一个实例变量。通常，我们不希望为每一个AbstractControl控件都创建一个实例变量。在没有实例变量的情况下，我们该如何在视图中引用FormControl呢？

我们可以改用myForm.controls属性。

code/forms/app/ts/forms/demo_form_with_validations_shorthand.ts

```
<input type="text"
       id="skuInput"
       placeholder="SKU"
       [formControl]="myForm.controls['sku']">
<div *ngIf="!myForm.controls['sku'].valid"
  class="ui error message">SKU is invalid</div>
<div *ngIf="myForm.controls['sku'].hasError('required')"
  class="ui error message">SKU is required</div>
```

通过这种方式，我们就不用被迫在组件类中显式定义实例变量来访问sku控件了。

5.6.2　自定义验证器

我们经常要写一些自定义验证器，下面来看看如何实现。

要明白如何实现自己的验证器，不妨看看Angular源代码中是如何实现Validators.required的：

```
export class Validators {
  static required(c: FormControl): StringMap<string, boolean> {
    return isBlank(c.value) || c.value == "" ? {"required": true} : null;
  }
```

一个验证器：

- 接收一个FormControl作为输入；
- 当验证失败时，会返回一个StringMap<string, boolean>对象，它的键是“错误代码”，值是true。

1. 编写验证器

假设我们的sku有特殊的验证需求，比如sku必须以123作为开始。我们写的验证器是这样的：

code/forms/app/ts/forms/demo_form_with_custom_validations.ts

```
function skuValidator(control: FormControl): { [s: string]: boolean } {
  if (!control.value.match(/^123/)) {
    return {invalidSku: true};
  }
}
```

当输入值（控件的值control.value）不是以123作为开始时，验证器会返回错误代码invalidSku。

2. 给FormControl分配验证器

现在要为FormControl添加验证，但是有一个小问题：sku已经有一个验证器了，怎样才能在同一个字段上添加多个验证器呢？

我们可以用Validators.compose来实现。

code/forms/app/ts/forms/demo_form_with_custom_validations.ts

```
constructor(fb: FormBuilder) {
  this.myForm = fb.group({
    'sku': ['', Validators.compose([
      Validators.required, skuValidator])]
  });
```

Validators.compose把两个验证器包装在一起，我们可以将其赋值给FormControl。只有当两个验证都合法时，FormControl才是合法的。

现在就能在视图中使用这个新的验证器了。

code/forms/app/ts/forms/demo_form_with_custom_validations.ts

```
<div *ngIf="sku.hasError('invalidSku')"
  class="ui error message">SKU must begin with <span>123</span></div>
```



注意，我们在本节中为每个FormControl都显式添加了实例变量。这意味着，在本节的视图中sku引用的是一个FormControl。

运行示例代码，你会注意到有一点很奇妙：当你在字段中输入一些内容时，满足了required验证，但违反了invalidSku验证。棒极了，这意味着我们可以对字段进行部分验证并显示相应的信息。

5.7 监听变化

到目前为止，我们只在提交表单时才调用onSubmit方法来获取表单的值。但我们也要经常监听控件的变化。

FormGroup和FormControl都带有EventEmitter（事件发射器），我们可以通过它来观察变化。



EventEmitter是一个**可观察**对象，符合“变化监听”规范。如果你对可观察对象的规范感兴趣，可以参见https://github.com/jhusain/observable-spec。

想监听控件的变化，我们要：

(1) 通过调用control.valueChanges访问到这个EventEmitter；

(2) 然后使用.subscribe方法添加一个监听器。

下面是一个例子。

code/forms/app/ts/forms/demo_form_with_events.ts

```
constructor(fb: FormBuilder) {
  this.myForm = fb.group({
    'sku': ['', Validators.required]
  });

  this.sku = this.myForm.controls['sku'];

  this.sku.valueChanges.subscribe(
    (value: string) => {
      console.log('sku changed to:', value);
    }
  );

  this.myForm.valueChanges.subscribe(
    (form: any) => {
      console.log('form changed to:', form);
    }
  );

}
```

在这里我们监听了两个事件：sku字段的变化和整个表单的变化。

我们传递了一个带有next键的对象（也可以传递其他键，但目前还不用关心它们）。next就是我们希望当值发生变化时被调用的函数。

如果在输入框中输入kj，就会在控制台中看到：

```
sku changed to:  k
form changed to:  Object {sku: "k"}
sku changed to:  kj
form changed to:  Object {sku: "kj"}
```

如你所见，每一次按键都会触发控件的变化，我们的可观察对象也会被触发。监听单个FormControl时，我们会得到一个值（例如kj）；而监听整个表单时，我们会得到一个包含键值对的对象（例如{sku: "kj"}）。

5.8　ngModel

ngModel是一个特殊的指令，它将模型绑定到表单中。ngModel的特殊之处在于它实现了双向绑定。相对于单向绑定来说，双向绑定更加复杂和难以推断。Angular通常的数据流向是单向的：自顶向下。但对于表单来说，双向绑定有时会更容易。



不要仅仅因为你以前在AngularJS中用过ng-model而急于使用ngModel，因为有很多避免使用双向绑定的理由。当然，ngModel确实用起来更方便，但要记住Angular已经不像AngularJS那样必须依赖双向绑定了。

下面对表单稍作修改：我们希望能输入产品名称productName。这次要用ngModel来保持组件实例和视图的同步。

首先，我们的组件定义类如下所示。

code/forms/app/ts/forms/demo_form_ng_model.ts

```
export class DemoFormNgModel {
  myForm: FormGroup;
  productName: string;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'productName': ['', Validators.required]
    });
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```

注意，我们只是简单地将productName: string存成了实例变量。

紧接着，我们在input标签上使用ngModel。

code/forms/app/ts/forms/demo_form_ng_model.ts

```
      <label for="productNameInput">Product Name</label>
      <input type="text"
             id="productNameInput"
             placeholder="Product Name"
             [formControl]="myForm.get('productName')"
             [(ngModel)]="productName">
```

注意，这里ngModel的语法很有意思：我们在ngModel属性上同时使用了()和[]。我们既使用了表示输入属性（@Input）的方括号[]，又使用了表示输出属性（@Output）的圆括号()，这就是双向绑定的标志。

另外还需要注意的是：我们仍然用formControl指定此input应该绑定到表单上的FormControl。这是因为ngModel只负责将input绑定到对象实例上，但FormControl的功能是与此独立的。由于我们还需要对这个值加以验证并把它作为表单的一部分提交上去，仍要保留formControl指令。

最后，我们把产品名称productName值显示在视图中。

code/forms/app/ts/forms/demo_form_ng_model.ts

```
    <div class="ui info message">
      The product name is: {{productName}}
    </div>
```

运行效果图如图5-4所示。

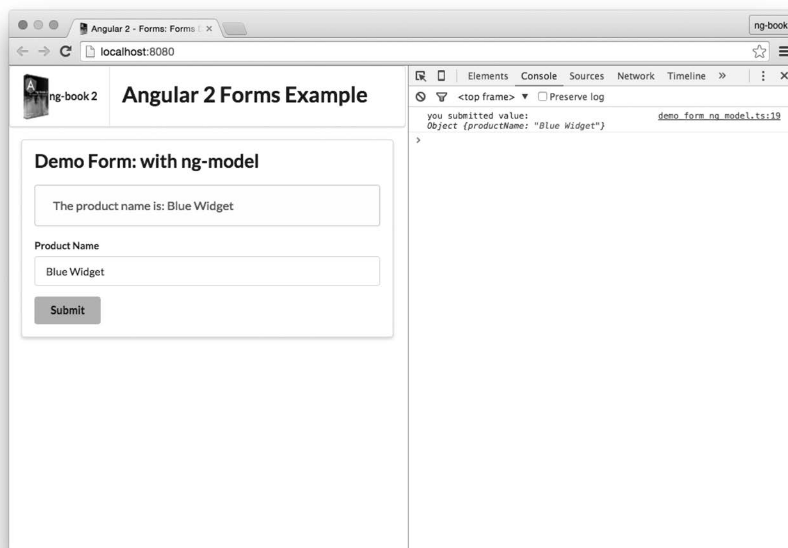


图5-4 带ngModel的演示表单

很简单吧！

5.9 总结

表单有很多零碎的知识，但Angular让它变得非常简明。只要我们掌握了如何使用FormGroup、FormControl和Validation，它就变得非常容易了！

第6章 HTTP

6.1 简介

Angular有自己的HTTP库，我们可以用它来调用外部的API。

当应用对外部服务器发出请求时，我们希望用户能继续与页面进行交互。也就是说，我们不希望页面在HTTP请求从外部服务器返回前一直失去响应。因此，我们的HTTP请求是异步的。

一直以来，处理异步代码比处理同步代码更加棘手。在JavaScript中，通常有3种处理异步代码的方式：

(1) 回调（callback）

(2) 承诺（promise）

(3) 可观察对象（observable）

在Angular中，处理异步代码的最佳方式就是使用可观察对象，所以我们会在本章中介绍这种方式。



关于RxJS和可观察对象：本章会涉及可观察对象的使用，但不会对其进行过多的解释。第10章会通过深入解析RxJS来讲解可观察对象。

在本章中，我们将：

(1) 展示一个`Http`的基本例子；

(2) 创建一个随敲随搜（search-as-you-type）组件用于搜索YouTube；

(3) 讨论`Http`库的API细节。



示例代码本章所用示例的完整代码可以在示例代码下的http文件夹中找到。文件夹中包含一个README.md文件，其中介绍了如何构建及运行项目。

在阅读本章时，最好尝试运行一下相关代码。请随意尝试，以深入了解这些代码的工作原理。

6.2　使用 @angular/http

HTTP在Angular中被拆分为一个单独的模块。这意味着你需要从@angular/http中导入一些常量。比如，我们通常会像下面这样导入@angular/http中的常量：

```
import { Http, Response, RequestOptions, Headers } from '@angular/http';
```

从@angular/http 中导入

在app.ts代码中，我们要导入HttpModule，这是一个便于使用的模块集合。

code/http/app/ts/app.ts

```
/*
 * Angular
 */
import {
  Component
} from '@angular/core';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { HttpModule } from '@angular/http';
```

我们把HttpModule作为依赖项，加入NgModule的imports列表之中。这样就可以把Http（和另外一些模块）导入组件之中。

code/http/app/ts/app.ts

```
@NgModule({
  declarations: [
    HttpApp,
    SimpleHTTPComponent,
    MoreHTTPRequests,
    YouTubeSearchComponent,
    SearchBox,
    SearchResultComponent
  ],
  imports: [
    BrowserModule,
    HttpModule // <--- right here
  ],
  bootstrap: [ HttpApp ],
  providers: [
    youTubeServiceInjectables
  ]
 })
class HttpAppModule {}
```

现在就可以把Http服务注入到组件中了。（实际上也可以用在任何使用依赖注入的地方。）

```
class MyFooComponent {
  constructor(public http: Http) {
  }

  makeRequest(): void {
    // do something with this.http ...
  }
}
```

6.3 基本请求

首先做的就是向jsonplaceholder API[1]发起一个简单的GET请求。

我们要做的是：

(1) 有一个调用makeRequest的button；

(2) makeRequest会调用http库向API发起一个GET请求；

(3) 当请求返回时，使用返回结果中的数据更新this.data。

该示例的截图如图6-1所示。

Basic Request

Make Request

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehende
rit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
}
```

图6-1　基本请求

6.3.1 构建 SimpleHTTPComponent 的@Component

首先要导入一些模块，然后指定@Component的selector。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
/*
 * Angular
 */
```

① http://jsonplaceholder.typicode.com

6

```
import {Component} from '@angular/core';
import {Http, Response} from '@angular/http';

@Component({
  selector: 'simple-http',
```

6.3.2　构建 SimpleHTTPComponent 的 template

然后构建视图。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
  template: `
  <h2>Basic Request</h2>
  <button type="button" (click)="makeRequest()">Make Request</button>
  <div *ngIf="loading">loading...</div>
  <pre>{{data | json}}</pre>
  `
```

要注意这里使用了ngIf指令。

模板中有三个有趣的部分：

(1) button

(2) 载入指示器

(3) data

我们将控制器中的makeRequest函数绑定到button的(click)上，稍后会对这个函数进行定义。

我们要向用户说明请求正在处理中，所以需要在变量loading为true的时候，使用ngIf来显示loading...。

data是一个Object。这里使用了json管道，这是一种非常棒的输出调试方式。把这段代码放进pre标签内就可以获得漂亮、易读的格式。

6.3.3　构建 SimpleHTTPComponent 控制器

我们先为SimpleHTTPComponent定义一个新的class。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
export class SimpleHTTPComponent {
  data: Object;
  loading: boolean;
```

现在，我们已经有了data和loading这两个实例变量。它们将分别用来存储API返回的数据值与表示加载状态。

然后定义constructor。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
constructor(private http: Http) {
}
```

constructor的方法体是空的，我们要注入一个关键模块Http。



需要记住，当我们在public http: Http中使用public关键字的时候，TypeScript会将http赋值给this.http。它是下面这种写法的简写：

```
// other instance variables here
http: Http;

constructor(http: Http) {
  this.http = http;
}
```


现在，我们就通过实现makeRequest函数来发起第一个HTTP请求。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
makeRequest(): void {
  this.loading = true;
  this.http.request('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}
```

当我们调用makeRequest时，首先要设置this.loading = true。这会在页面上显示载入指示器。

发起HTTP请求的方式非常简明：调用this.http.request并传入URL作为GET请求的参数。

http.request会返回一个Observable对象。我们可以使用subscribe订阅变化（类似于在一个promise上使用then）。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
  this.http.request('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
```

当http.request（从服务器）返回一个流时，它就会发出一个Response对象。我们用json方法提取出响应体并解析成一个Object，然后将这个Object赋值给this.data。

只要我们得到了响应，就不会再加载任何东西了，所以这里需要设置this.loading = false。



.subscribe同样可以处理失败和流完结的情况，只要分别在第二和第三个参数中传入一个函数就可以了。对于一个产品级应用来说，处理这两种情况是个好主意。当请求失败（即流中发生错误）的时候，this.loading也应当被设置为false。

6.3.4　完整的 SimpleHTTPComponent

下面就是完整的SimpleHTTPComponent。

code/http/app/ts/components/SimpleHTTPComponent.ts

```
/*
 * Angular
 */
import {Component} from '@angular/core';
import {Http, Response} from '@angular/http';

@Component({
  selector: 'simple-http',
  template: `
  <h2>Basic Request</h2>
  <button type="button" (click)="makeRequest()">Make Request</button>
  <div *ngIf="loading">loading...</div>
  <pre>{{data | json}}</pre>
`
})
export class SimpleHTTPComponent {
  data: Object;
  loading: boolean;

  constructor(private http: Http) {
  }

  makeRequest(): void {
    this.loading = true;
    this.http.request('http://jsonplaceholder.typicode.com/posts/1')
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
}
```

6.4　编写 YouTubeSearchComponent

上一个例子是从代码中获取API服务器上数据的最简方式。现在我们要尝试构建一个更复杂的例子。

在这一节里，我们会打造一个随着输入搜索YouTube的组件。当搜索结果返回时，通过一个列表来展示每一个视频的缩略图、描述和链接。

搜索cats playing ipads时的屏幕截图如图6-2所示。

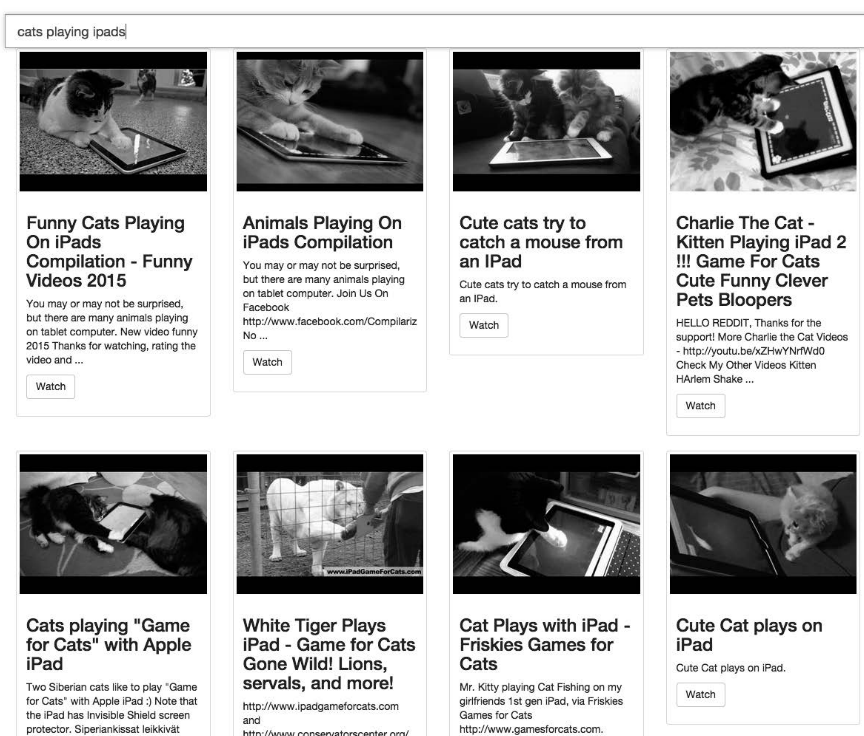


图6-2　能让我的猫咪写Angular吗

在这个例子中，我们要实现下列功能：

(1) 一个`SearchResult`对象，用于存放每条搜索结果；

(2) 一个`YouTubeService`服务，用于管理向YouTube的API发出的请求并将结果转成一个`SearchResult[]`流；

(3) 一个`SearchBox`组件，用于根据用户输入内容调用YouTube服务；

(4) 一个`SearchResultComponent`组件，用于渲染具体的`SearchResult`结果；

(5) 一个`YouTubeSearchComponent`组件，封装整个YouTube搜索功能并渲染结果列表。

下面逐一处理每个部分。



Patrick Stapleton维护着一个非常棒的代码仓库angular2-webpack-starter[1]。里面有使用RxJS实现搜索GitHub仓库时自动补全的示例。本节中的一些想法就是受这个示例的启发。它是个包含各种示例的酷炫项目，也许你该看一下。

6.4.1　编写 SearchResult

我们先从编写一个基本的SearchResult类开始。这个类为我们存储搜索结果中一些感兴趣的字段提供了一种便捷的方式。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id            || null;
    this.title           = obj && obj.title         || null;
    this.description     = obj && obj.description   || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl  || null;
    this.videoUrl        = obj && obj.videoUrl      ||
                               `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

这里使用obj?: any方式来模拟关键词参数。我们可以创建一个新的SearchResult并且只传入一个包含指定键的对象。

唯一要特别指出的是，我们在构造videoUrl时使用了硬编码的URL格式。你也可以将其重构为一个根据多个参数来生成路径的函数，或者直接在视图中使用视频的id来构造URL。

6.4.2　编写 YouTubeService

1. API

在这个例子中，我们将使用YouTube第3版搜索 API[2]。

① https://github.com/angular-class/angular2-webpack-starter
② https://developers.google.com/youtube/v3/docs/search/list



为了使用这个API，你需要一个API密钥。我们已经在示例代码中包含了一个可供大家使用的API密钥。尽管如此，当你读到这里的时候，可能发现这个密钥已经超过了使用频率限制。如果是这样的话，你就需要去生成一个自己的密钥了。

要生成自己的密钥，可以查看文档：https://developers.google.com/youtube/registering_an_application#Create_API_Keys。为了简单起见，我已经注册了一个服务器密钥；如果你要将你的JavaScript代码放到线上，那么还需要一个浏览器密钥。

我们将为`YouTubeService`设置两个用来表示API密钥和API URL的常量：

```
let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

最后，还要测试一下应用。我们并不希望在产品环境下进行测试，而是希望测试预生产或开发阶段的API。

为了解决这个环境配置问题，我们就要让这些常量可被注入。

为什么要注入这些常量，而不是像平常那样直接使用呢？这是因为只要让这些常量可被注入，我们就能：

(1) 让代码在部署的时候根据所选环境注入正确的常量；

(2) 在测试期更容易替换要注入的值。

通过注入这些值，我们将获得更多的灵活性。

为了让这些值可被注入，我们使用`{ provide: ... , useValue: ... }`语法。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
export var youTubeServiceInjectables: Array<any> = [
  {provide: YouTubeService, useClass: YouTubeService},
  {provide: YOUTUBE_API_KEY, useValue: YOUTUBE_API_KEY},
  {provide: YOUTUBE_API_URL, useValue: YOUTUBE_API_URL}
];
```

这里我们指定，要把`YOUTUBE_API_KEY`的值绑定到可被注入的`YOUTUBE_API_KEY`上。（`YOUTUBE_API_URL`也一样，稍后还们还将定义`YouTubeService`。）

也许你还记得，为了在本应用中进行依赖注入，我们需要将其放入`NgModule`的`providers`里。因为这里导出了`youTubeServiceInjectables`，所以就能在app.ts中使用它了。

```
// http/app.ts
import { HttpModule } from '@angular/http';
import { youTubeServiceInjectables } from "components/YouTubeSearchComponent";

// ...
```

```
// further down
// ...

@NgModule({
  declarations: [
    HttpApp,
    // others ....
  ],
  imports: [ BrowserModule, HttpModule ],
  bootstrap: [ HttpApp ],
  providers: [
    youTubeServiceInjectables // <--- right here
  ]
})
class HttpAppModule {}
```

现在我们使用注入（来自youTubeServiceInjectables的）YOUTUBE_API_KEY的方式来代替直接使用变量。

2. YouTubeService构造函数

我们通过编写一个class并使用@Injectable对其进行注解来创建YouTubeService。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeService {
  constructor(private http: Http,
              @Inject(YOUTUBE_API_KEY) private apiKey: string,
              @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }
```

我们在constructor中注入三样东西：

(1) Http

(2) YOUTUBE_API_KEY

(3) YOUTUBE_API_URL

这里要注意，我们使用这三个参数创建实例变量。这意味着可以分别通过this.http、this.apiKey和this.apiUrl来访问它们。

还要注意，我们使用@Inject(YOUTUBE_API_KEY)进行显式注入。

3. YouTubeService搜索

下一步，我们来实现search函数。search传入一个要查询的string并返回一个会发出SearchResult[]流的Observable。换句话说，它发出的每个条目都是一个SearchResult数组。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  search(query: string): Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    let queryUrl: string = `${this.apiUrl}?${params}`;
```

这里使用了手动的方式来构造queryUrl。我们简单地将查询参数放入params变量之中。（你可以查阅搜索API文档[1]了解每个值的含义。）

然后将apiUrl与params拼接起来作为queryUrl。

现在就有了一个可以用来发起请求的queryUrl了。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  search(query: string): Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    let queryUrl: string = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
  }
```

我们要获取http.get的返回值，并用map来从请求中获取Response。这里使用.json()从response中提取返回体并同时实例化成一个对象。然后遍历每一个项目并将其转换成一个SearchResult。

① https://developers.google.com/youtube/v3/docs/search/list



如果你想看看item的原始值，可以取消对console.log的注释，然后在浏览器的开发者控制台检查输出的值。



注意，这里调用了(<any>response.json()).items。这是在干什么？这是在告诉TypeScript，我们并不想在这里进行严格的类型检查。

当我们使用JSON API时，通常并没有API响应体的类型定义信息，所以TypeScript不知道返回的Object中会有一个items键。因此，编译器会在这里出问题。

我们也可以调用response.json()["items"]并将其转换成一个Array类型，但是这里（以及创建SearchResult时）将其作为any类型来使用会更加简洁，只是牺牲了一点类型检查的严格性。

4. YouTubeService的完整代码

这里是YouTubeService的完整代码。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * YouTubeSearchComponent is a tiny app that will autocomplete search YouTube.
 */

import {
  Component,
  Injectable,
  OnInit,
  ElementRef,
  EventEmitter,
  Inject
} from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs';

/*
 This API key may or may not work for you. Your best bet is to issue your own
 API key by following these instructions:
 https://developers.google.com/youtube/registering_an_application#Create_API_Ke\
ys

 Here I've used a **server key** and make sure you enable YouTube.

 Note that if you do use this API key, it will only work if the URL in
 your browser is "localhost"
*/
export var YOUTUBE_API_KEY: string = 'AIzaSyDOfT_BO81aEZScosfTYMruJobmpjqNeEk';
export var YOUTUBE_API_URL: string = 'https://www.googleapis.com/youtube/v3/sear\
ch';
```

```
let loadingGif: string = ((<any>window).__karma__) ? '' : require('images/loadin\
g.gif');

class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id             || null;
    this.title           = obj && obj.title          || null;
    this.description     = obj && obj.description    || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl   || null;
    this.videoUrl        = obj && obj.videoUrl       ||
                               `https://www.youtube.com/watch?v=${this.id}`;
  }
}

/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeService {
  constructor(private http: Http,
              @Inject(YOUTUBE_API_KEY) private apiKey: string,
              @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }

  search(query: string): Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    let queryUrl: string = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
  }
}
```

```
export var youTubeServiceInjectables: Array<any> = [
  {provide: YouTubeService, useClass: YouTubeService},
  {provide: YOUTUBE_API_KEY, useValue: YOUTUBE_API_KEY},
  {provide: YOUTUBE_API_URL, useValue: YOUTUBE_API_URL}
];

/**
 * SearchBox displays the search box and emits events based on the results
 */

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBox implements OnInit {
  loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>();

  constructor(private youtube: YouTubeService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.next(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.next(false);
          this.results.next(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.next(false);
        },
        () => { // on completion
          this.loading.next(false);
        }
      );

  }
}
```

```
@Component({
  inputs: ['result'],
  selector: 'search-result',
  template: `
    <div class="col-sm-6 col-md-3">
      <div class="thumbnail">
        <img src="{{result.thumbnailUrl}}">
        <div class="caption">
          <h3>{{result.title}}</h3>
          <p>{{result.description}}</p>
          <p><a href="{{result.videoUrl}}"
                class="btn btn-default" role="button">
                Watch</a></p>
        </div>
      </div>
    </div>
  `
})
export class SearchResultComponent {
  result: SearchResult;
}

@Component({
  selector: 'youtube-search',
  template: `
  <div class='container'>
      <div class="page-header">
        <h1>YouTube Search
          <img
            style="float: right;"
            *ngIf="loading"
            src='${loadingGif}' />
        </h1>
      </div>

      <div class="row">
        <div class="input-group input-group-lg col-md-12">
          <search-box
             (loading)="loading = $event"
             (results)="updateResults($event)"
             ></search-box>
        </div>
      </div>

      <div class="row">
        <search-result
          *ngFor="let result of results"
          [result]="result">
        </search-result>
      </div>
  </div>
  `
})
export class YouTubeSearchComponent {
```

```
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

6.4.3　编写 SearchBox

SearchBox组件在应用中扮演着关键的角色：它是UI与YouTubeService的中间连接层。

SearchBox将会：

(1) 观察input的keyup事件，并向YouTubeService提交搜索；

(2) 在正在加载（或者不再加载）时，触发一个loading事件；

(3) 在获取到新的结果时，触发一个results事件。

1. 定义SearchBox的@Component

我们来定义SearchBox的@Component。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * SearchBox displays the search box and emits events based on the results
 */

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
```

我们之前已经见过很多次selector了：它允许我们创建<search-box>标签。

outputs指定了将从组件中触发的事件，也就是可以在视图中使用(output)="callback()"语法以侦听组件中的事件。例如，下面是我们将在视图中使用search-box标签的方式：

```
<search-box
 (loading)="loading = $event"
 (results)="updateResults($event)"
  ></search-box>
```

在这个例子中，当SearchBox组件触发一个loading事件时，我们要设置父上下文中的loading变量。同样，当SearchBox组件触发results事件时，我们将会调用父上下文中的updateResults()函数。

我们在@Component的配置当中简要地用"loading"和"results"字符串指定事件的名称。在这个例子中，每个事件都会有一个对应的EventEmitter作为控制器类的实例变量。稍后就会实

现它们。

目前，要记住@Component就像是组件的公共API，所以这里只需要指定事件的名称，稍后再来看EventEmitter的具体实现。

2. 定义SearchBox的template

我们的template很简明。这里只有一个 input 标签。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * SearchBox displays the search box and emits events based on the results
 */

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
```

3. 定义SearchBox控制器

SearchBox控制器是一个新类。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
export class SearchBox implements OnInit {
  loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>();
```

我们通过implements OnInit让这个类实现对应的接口，这么做是因为需要使用生命周期中ngOnInit的回调。如果一个类声明implements OnInit，那么ngOnInit函数会在首次变化检查后调用。

ngOnInit是进行初始化工作的理想地方（相对于constructor），因为组件的各个输入参数在constructor中仍然是不可用的。

- 定义SearchBox控制器的constructor

我们来看一下SearchBox的constructor。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  constructor(private youtube: YouTubeService,
              private el: ElementRef) {
  }
```

我们在constructor中注入：

(1) YouTubeService

(2) 此组件所附着的元素el

其中的el是一个ElementRef类型的对象，此类型是Angular对原生元素的一个包装。

我们将注入的两个值作为实例变量。

- 定义SearchBox控制器ngOnInit

在输入框中，我们想要监视keyup事件。问题是，如果在每一次keyup后都直接进行搜索，可能效果并不好。我们可以用三种方式来提升用户体验：

(1) 过滤掉空白与过短的查询；

(2) 消除输入的“抖动”，也就是我们不希望每一个字符发生改变时都进行搜索，而是在用户完成输入并暂停一小段时间后再进行搜索；

(3) 当用户进行新的搜索时，抛弃旧的搜索内容。

我们可以手动绑定keyup，并在每次keyup事件触发时调用一个函数，然后在其中实现字符过滤与抖动消除。不过我们有一种更好的方式：让keyup事件成为一个可观察流。

RxJS提供了一种使用Rx.Observable.fromEvent的方式来监听一个元素上的事件。我们可以像下面这样使用它。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
ngOnInit(): void {
  // convert the `keyup` event into an observable stream
  Observable.fromEvent(this.el.nativeElement, 'keyup')
```

要注意在fromEvent里面：

- 第一个参数是this.el.nativeElement（组件附着的原生DOM元素）；
- 第二个参数是字符串'keyup'，代表的是将要被转换成流的事件名称。

借助流的魔力，我们可以把它转换成SearchResult。下面来分步看看。

有了keyup事件的流，就能把多个方法串联起来。接下来我们会在流上串联一些转换流的函数，并在最后展示整个示例。

首先，我们要从input标签中提取输入值：

```
.map((e: any) => e.target.value) // extract the value of the input
```

上面的代码表示，映射每一个keyup事件，然后找到它的目标（e.target，也就是input元素）并取出value。这意味着这个流现在变成了一个字符串流。

下一步：

```
.filter((text: string) => text.length > 1)
```

filter表示该流在长度小于1的时候不会发送任何搜索字符串。如果你还希望忽略较短的搜索字符串，可以把这个值改大一点。

```
.debounceTime(250)
```

debounceTime表示我们会忽略触发间隔小于250 ms的请求。也就是说，我们不会去搜索每一次键入的内容。只有在用户暂停输入一小段时间后才会触发搜索。

```
.do(() => this.loading.next(true))        // enable loading
```

在流上使用do方法可以在流中对每个事件执行函数，但是这种方式不会改变流中的任何数据。这是因为已经获取到了具有足够长度并消除了输入抖动的搜索字符串，所以要在页面上显示loading。

this.loading是一个EventEmitter。我们通过发射true作为下一个事件来“开启”loading。我们通过调用next来在EventEmitter上发射数据。编写的this.loading.next(true)代表在loading这个EventEmitter上发射一个true事件。当监听此组件上的loading事件时，$event的值现在会被设置为true（稍后会深入探讨使用$event）。

```
.map((query: string) => this.youtube.search(query))
.switch()
```

在每一个触发的查询上使用.map以执行搜索。使用switch表示“除了最近的一次，忽略所有搜索事件”。这就是说，如果有一个新的搜索进来，我们就使用这个最新的并丢弃掉其他搜索。



熟悉Reactive的专家对此一定不会陌生。你也可以在RxJS的文档[①]中找到关于switch方法的更加具体的定义。

每当进入query时，都将对YouTubeService进行一次搜索（search）。

把这些串联在一起，结果如下所示。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
     .filter((text: string) => text.length > 1) // filter out if empty
     .debounceTime(250)                         // only once every 250ms
     .do(() => this.loading.next(true))         // enable loading
     // search, discarding old events if new input comes in
     .map((query: string) => this.youtube.search(query))
     .switch()
     // act on the return of the search
     .subscribe(
```

① https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/switch.md

因为RxJS的API数量众多，所以看起来会有些吓人。尽管如此，我们使用简单的几行代码就实现了一个极为复杂的事件处理流！

因为是在调用YouTubeService，所以我们的流现在是一个SearchResult[]流了。这时可以订阅（subscribe）这个流，并执行相应的操作。

subscribe接收三个参数：onSuccess、onError和onCompletion。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.next(false);
          this.results.next(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.next(false);
        },
        () => { // on completion
          this.loading.next(false);
        }
      );

  }
```

第一个参数指定了当流触发一个正常事件时将会执行的操作。这里我们会在这两个EventEmitter上触发一个事件：

(1) 调用this.loading.next(false)，表示停止加载；

(2) 调用this.results.next(results)，会触发一个包含结果列表数据的事件。

第二个参数指定了当流出现错误时将会执行的操作。这里我们只设置 this.loading.next(false) 并记录下错误。

第三个参数指定了当流结束时将会执行的操作。这里依然会触发结束加载的事件。

4. SearchBox组件的完整代码

以下是SearchBox组件的完整代码。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * SearchBox displays the search box and emits events based on the results
 */

@Component({
  outputs: ['loading', 'results'],
  selector: 'search-box',
  template: `
```

```
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBox implements OnInit {
  loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResult[]>();

  constructor(private youtube: YouTubeService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.next(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.next(false);
           this.results.next(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.next(false);
        },
        () => { // on completion
          this.loading.next(false);
        }
      );

  }
}
```


6.4.4 编写 SearchResultComponent

之前的SearchBox相当复杂。现在来处理一个简单得多的组件：SearchResultComponent（如图6-3所示）。SearchResultComponent的作用就是渲染一个SearchResult。

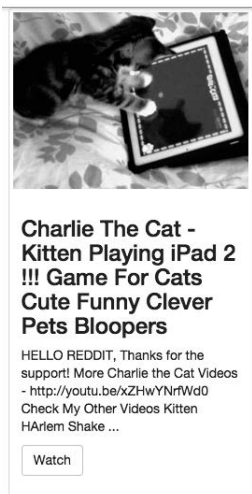


图6-3　单一搜索结果组件

这里没有什么新东西，所以直接完整地列出来。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
@Component({
  inputs: ['result'],
  selector: 'search-result',
  template: `
   <div class="col-sm-6 col-md-3">
      <div class="thumbnail">
        <img src="{{result.thumbnailUrl}}">
        <div class="caption">
          <h3>{{result.title}}</h3>
          <p>{{result.description}}</p>
          <p><a href="{{result.videoUrl}}"
                class="btn btn-default" role="button">
                Watch</a></p>
        </div>
      </div>
    </div>
  `
})
export class SearchResultComponent {
  result: SearchResult;
}
```

有以下几点需要关注：

- @Component只有一个result输入参数，可以通过它把SearchResult赋值给组件；
- template里有标题、描述以及视频的缩略图，并通过一个按钮链接到视频上；
- SearchResultComponent在其实例中使用result变量存储SearchResult。

6.4.5　编写 YouTubeSearchComponent

我们要实现的最后一个组件就是YouTubeSearchComponent。这个组件最终会将所有东西组织在一起。

1. YouTubeSearchComponent的@Component

code/http/app/ts/components/YouTubeSearchComponent.ts

```
@Component({
  selector: 'youtube-search',
```

@Component注解很容易理解：使用名为youtube-search的selector。

2. YouTubeSearchComponent控制器

在讨论template之前，需要先看一下YouTubeSearchComponent控制器。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
export class YouTubeSearchComponent {
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

这个组件拥有一个实例变量：SearchResult数组类型的results。

我们还定义了一个函数：updateResults。updateResults直接把SearchResult[]的新值赋给this.results。

results和updateResults都会在template中用到。

3. YouTubeSearchComponent的template

我们的视图需要做三件事：

(1) 在加载时，显示加载指示器；

(2) 监听search-box上的事件；

(3) 显示搜索结果。

之后来看一下template。构建基本结构并在头部的旁边显示表示“正在加载”的gif动画。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  template: `
  <div class='container'>
      <div class="page-header">
        <h1>YouTube Search
```

```
      <img
        style="float: right;"
        *ngIf="loading"
        src='${loadingGif}' />
    </h1>
  </div>
```



注意，img的src属性为${loadingGif}，loadingGif变量来自于程序前面的require语句。这里使用了webpack的图像文件加载功能。如果你想探究其工作原理，可以看一下本章示例代码中的webpack配置，或者下载image-webpack-loader项目[①]。

因为只有当loading为真时，才需要显示加载图像，所以要用ngIf来实现这个功能。

接下来，看看使用search-box的地方。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
<div class="row">
  <div class="input-group input-group-lg col-md-12">
    <search-box
       (loading)="loading = $event"
       (results)="updateResults($event)"
        ></search-box>
  </div>
```

值得关注的是将results输出结果绑定到loading的方式。注意我们在这里使用了(output)="action()"语法。

对于loading输出，运行loading = $event表达式。$event会被EventEmitter发出的事件值替换掉。也就是说，当我们调用SearchBox组件中的this.loading.next(true)时，$event的值将会是true。

同样，对于results输出，每当一组新的结果发出时，都会调用updateResults()函数。这样就能实现更新组件中results实例变量值的效果。

最后，我们要在组件中获取results列表，并为每个组件渲染一个search-result。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
  <div class="row">
    <search-result
      *ngFor="let result of results"
      [result]="result">
    </search-result>
  </div>
</div>
```

① https://github.com/tcoopman/image-webpack-loader

4. YouTubeSearchComponent的完整代码

这里是YouTubeSearchComponent的完整代码。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
@Component({
  selector: 'youtube-search',
  template: `
  <div class='container'>
      <div class="page-header">
        <h1>YouTube Search
          <img
            style="float: right;"
            *ngIf="loading"
            src='${loadingGif}' />
        </h1>
      </div>

      <div class="row">
        <div class="input-group input-group-lg col-md-12">
          <search-box
             (loading)="loading = $event"
             (results)="updateResults($event)"
              ></search-box>
        </div>
      </div>

      <div class="row">
        <search-result
          *ngFor="let result of results"
          [result]="result">
        </search-result>
      </div>
  </div>
  `
})
export class YouTubeSearchComponent {
  results: SearchResult[];

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

好了！这样我们就实现了一个针对YouTube视频的随敲随搜功能！如果你还不太明白，可以尝试执行示例代码。

6.5 @angular/http API

当然，到目前为止发起的所有HTTP请求都是简单的GET请求。知晓如何发起其他类型的请求也很重要。

6.5.1 发起一个 POST 请求

使用@angular/http发起POST请求与发起GET请求非常类似，仅仅多了一个额外的参数：请求体。

jsonplaceholder API[1]同样提供了一个URL，可供测试POST请求。现在就来试一下。

code/http/app/ts/components/MoreHTTPRequests.ts

```
makePost(): void {
  this.loading = true;
  this.http.post(
    'http://jsonplaceholder.typicode.com/posts',
    JSON.stringify({
      body: 'bar',
      title: 'foo',
      userId: 1
    }))
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}
```

在第二个参数中，使用JSON.stringify将Object转换为一个JSON字符串。

6.5.2 PUT/PATCH/DELETE/HEAD

还有其他一些常见的HTTP请求，也是用类似的方式进行调用。

- http.put和http.patch分别用于PUT和PATCH请求，并且它们都带有一个URL和一个请求体。
- http.delete和http.head分别用于DELETE和HEAD请求，并且都带有一个URL（没有请求体）。

下面展示了如何发起一个DELETE请求。

code/http/app/ts/components/MoreHTTPRequests.ts

```
makeDelete(): void {
  this.loading = true;
```

① http://jsonplaceholder.typicode.com

```
  this.http.delete('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}
```

6.5.3 RequestOptions

目前我们覆盖到的所有http方法还带有一个可选的末位参数：RequestOptions。RequestOptions 对象封装了：

- method
- headers
- body
- mode
- credentials
- cache
- url
- search

比如，我们可以用X-API-TOKEN这样一个特殊的请求头来创建GET请求。

code/http/app/ts/components/MoreHTTPRequests.ts

```
makeHeaders(): void {
  let headers: Headers = new Headers();
  headers.append('X-API-TOKEN', 'ng-book');

  let opts: RequestOptions = new RequestOptions();
  opts.headers = headers;

  this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
    .subscribe((res: Response) => {
      this.data = res.json();
    });
}
```

6.6 总结

@angular/http非常灵活并且广泛适用于各种API。

@angular/http的一个强大特性就是支持模拟后台。这一点在测试中非常有用。想了解更多关于测试HTTP的内容，请参见第15章。

第 7 章

路由

7

在Web开发中，路由是指将应用划分成多个分区，通常是按照从浏览器的URL衍生出来的规则进行分割。

例如，访问一个网站的/路径时，我们有可能正在访问该网站的home路由；又例如，访问/about时，我们想要渲染的是关于页面；等等。

7.1 为什么需要路由

在应用程序中定义路由非常有用，因为我们可以：

- 将应用程序划分为多个分区；
- 维护应用程序的状态；
- 基于某些规则保护应用分区。

假设我们正在开发类似于前面描述的库存应用程序。

第一次访问该应用程序时，首先看到的可能是搜索表单，用来输入搜索关键词并获得匹配的产品列表。

然后，单击某产品可以访问该产品的详细信息页面。

因为我们的应用程序是客户端，所以变换“页面”并不一定要更改URL。但是值得考量的是，如果为所有页面使用同样的URL，会有什么后果呢？

- 刷新页面后，无法保留你在应用中的位置。
- 不能为页面添加书签，方便以后返回相同的页面。
- 无法与他人分享当前页面的URL。

反过来看，使用路由能让我们定义URL字符串，指定用户在应用中的位置。

在库存的例子中，我们可以为每个任务定义一系列不同的路由配置，如下所示。

- 最初的根URL可能是http://our-app/。当访问该路径时，我们可能被重定向到home路由：

http://our-app/home。

- 当访问“About Us”区域时，URL地址可能变为http://our-app/about。这样，如果我们将http://our-app/about发给其他用户，他们会看到相同的页面。

7.2 客户端路由的工作原理

也许你以前曾经编写过服务端的路由代码（这并不是完成本章的条件）。通常，在服务器端负责路由的情况下，收到HTTP请求后，服务器会根据收到的URL来运行相应的控制器。

例如，在Express.js[1]中，可以这样实现：

```
var express = require('express');
var router = express.Router();

// define the about route
router.get('/about', function(req, res) {
  res.send('About us');
});
```

在Ruby on Rails[2]中，可以这样实现：

```
# routes.rb
get '/about', to: 'pages#about'

# PagesController.rb
class PagesController < ActionController::Base
  def about
    render
  end
end
```

每种框架的模式各不相同，但是在上面两种情况中，你都有一个服务器。它接收一个请求，并路由到一个控制器。该控制器根据路径和参数执行特定的任务。

客户端路由在概念上很相似，但是实施方法不同。在客户端路由的情况下，每次URL发生变化时，不一定会向服务器发送请求。我们把Angular应用叫作单页应用程序（single page app，SPA），因为服务器只提供一个页面，负责渲染各种页面的是JavaScript。

那么，如何才能在JavaScript代码中设定各个路由呢？

7.2.1 初级阶段：使用锚标记

在初级阶段，客户端路由使用了一个巧妙的方法：它不使用指向各种页面的客户端URL，而

① http://expressjs.com/guide/routing.html

② http://rubyonrails.org/

是使用锚标记。

可能你已经知道，锚标记的传统作用是直接链接到所在网页的其他位置，并让浏览器滚动到定义该锚标记元素所在的位置。例如，如果在HTML页面中定义这样的锚标记：

```
<!-- ... lots of page content here ... -->
<a name="about"><h1>About</h1></a>
```

当访问http://something/#about这个URL时，浏览器将直接跳到这个定义about锚标记的H1标签。

SPA应用客户端框架使用的方式是：将锚标记作为路径来格式化，用它们代表应用程序的路由。

例如，SPA应用的about路由可能是http://something/#/about。这就是所谓的基于锚点标记的路由（hash-based routing）。

这个方法巧妙的地方在于，它看起来像一个“普通”的URL，因为它以锚标记和斜杠开头（/about）。

7.2.2　进化：HTML5 客户端路由

随着HTML5的引入，浏览器获得了新的能力：在不需要新请求的情况下，允许在代码中创建新的浏览器记录项并显示适当的URL。

这是利用history.pushState方法来实现的，该方法允许JavaScript控制浏览器的导航历史。

因此，现代框架可以不依赖锚标记方法来进行路由导航，而是依赖pushState在无需重新加载的情况下控制浏览器历史。



AngularJS注意事项：AngularJS应用已经可以使用这种路由方法了，但是需要使用$locationProvider.html5Mode(true)来特别启用。

在Angular中，HTML5路由是默认的模式。在本章后面，我们将讲解如何从HTML5模式退回到老的锚标记模式。



使用HTML5路由模式的时候，需要注意以下两点。

(1) 并非所有的浏览器都支持HTML5路由模式，所以如果需要支持老版浏览器，你可能会被迫使用基于锚点标记的路由模式。

(2) 服务器必须支持基于HTML5的路由。

为什么服务器必须要支持基于HTML5路由？我们将在后面深入讨论。

7.3 编写第一个路由配置



Angular文档建议使用HTML5路由模式[1]，但是鉴于上一节提到的种种挑战，我们会在例子中使用基于锚点标记的路由模式进行简化。

在Angular中，我们通过将路径映射到处理它们的组件来配置路由。

我们来创建一个有多种路由的小型应用程序。在这个例子应用程序中，我们将有三种路由：

- 主页，使用/#/home路径；
- 关于页面，使用/#/about路径；
- 联系我们页面，使用/#/contact路径；

最后，当用户访问根路径（/#/）时，重定向到主页路径。

7.4 Angular 路由的组成部件

7

我们使用三种主要部件来配置Angular路由。

- `Routes`：描述了应用程序支持的路由配置。
- `RouterOutlet`：这是一个“占位符”组件，用于告诉Angular要把每个路由的内容放在哪里。
- `RouterLink`指令：用于创建各种路由链接。

让我们来进一步讨论它们。

7.4.1 导入

为了使用Angular的路由器，首先从`@angular/router`库中导入一些常量。

code/routes/basic/app/ts/app.ts

```
import {
  RouterModule,
  Routes
} from '@angular/router';
```

现在，我们可以开始定义路由器配置了。

7.4.2 路由配置

为了定义应用的路由配置，首先创建一个`Routes`配置，然后使用`RouterModule.forRoot`

① https://angular.io/docs/ts/latest/guide/router.html#!#browser-url-styles

(routes)来为应用程序提供使用路由器必需的依赖。

code/routes/basic/app/ts/app.ts

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
];
```

注意关于路由配置的以下事项。

- path：指定了该路由要处理的URL路径。
- component：用于连接当前路由路径与处理该路由的组件。
- redirectTo：一个可选选项，用于将当前路径重定向到另一个已知路由。

综上所述，路由配置的目的是指定组件要处理的路径。

重定向

在路由定义中使用redirectTo是在告诉路由器，在访问该路由的path时，我们想让浏览器重定向到另一个路由。

在上面的示例代码中，如果访问http://localhost:8080/#/根路径，我们将被重定向到home路由。

另一个例子是contactus路由。

code/routes/basic/app/ts/app.ts

```
  { path: 'contactus', redirectTo: 'contact' },
```

在这种情况下，如果访问http://localhost:8080/#/contactus这个URL，那么浏览器将重定向到/contact。



示例代码　本节例子的完整代码可以在示例代码中的routes/basic目录中找到。查阅README.md文件，了解构建和运行本例的步骤。
路由需要多种导入声明，我们在下面的例子中不会逐一列出全部的导入声明。但是，我们为每个例子列出了源文件的文件名和行号。如果你遇到不知道如何导入某些类的问题，请使用编辑器打开代码文件并查看完整代码。
在阅读本节的同时，尝试运行代码并随意发挥可以获得更加深刻的认识。

7.4.3　安装路由配置

现在有了路由配置routes，我们需要安装它。为了在应用中使用路由配置，首先要对NgModule进行两项修改：

(1) 导入RouterModule；

(2) 在NgModule中的imports数组里使用RouterModule.forRoot(routes)来安装路由配置。

下面是为本应用在NgModule中配置的路由。

code/routes/basic/app/ts/app.ts

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
];

@NgModule({
  declarations: [
    RoutesDemoApp,
    HomeComponent,
    AboutComponent,
    ContactComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy }
  ]
})
class RoutesDemoAppModule {}

platformBrowserDynamic().bootstrapModule(RoutesDemoAppModule)
  .catch((err: any) => console.error(err));
```

7.4.4　使用<router-outlet>调用 RouterOutlet 指令

当路由发生变化时，我们希望保留外部“布局”模板，只用路由的组件替换页面的“内部”。

为了指定Angular在页面的什么地方渲染各种路由的内容，我们使用RouterOutlet指令。

组件的模板中指定了一些div结构、导航部分和一个名为router-outlet的指令。

router-outlet元素标示了各个路由组件的内容应该在哪里被渲染。



我们可以在模板中使用router-outlet指令，因为已经在NgModule中导入了RouterModule。

下面是应用中用于承载导航的组件及其模板。

code/routes/basic/app/ts/app.ts

```
@Component({
  selector: 'router-app',
  template: `
  <div>
    <nav>
      <a>Navigation:</a>
      <ul>
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['about']">About</a></li>
        <li><a [routerLink]="['contact']">Contact Us</a></li>
      </ul>
    </nav>

    <router-outlet></router-outlet>
  </div>
  `
})
class RoutesDemoApp {
}
```

仔细查看上面模板的内容，你将发现router-outlet元素在导航目录的正下方。当访问/home时，这里便是HomeComponent模板被渲染的地方。其他组件的渲染位置也是一样的。

7.4.5　使用[routerLink]调用 routerLink 指令

我们现在知道路由组件的模板将在哪里被渲染，那么如何才能让Angular导航到一个指定路由呢？

我们可以尝试使用纯HTML，像这样直接链接到路由：

```
<a href="/#/home">Home</a>
```

但是如果这样做，点击这个链接将触发页面重载，而这是开发单页应用时要杜绝的。

要解决这个问题，Angular提供了一个方案，可以在不重载页面的情况下链接路由：使用routerLink指令。

该指令允许你使用特殊的语法写链接。

code/routes/basic/app/ts/app.ts

```
      <a>Navigation:</a>
      <ul>
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['about']">About</a></li>
        <li><a [routerLink]="['contact']">Contact Us</a></li>
      </ul>
```

我们可以在左手边看到[routerLink]，它将该指令用于当前元素（<a>标签）。

在右手边是一组数组，它的第一个元素是路由的路径，比如"['home']"或者"['about']"，用来指定点击该元素时应该导航到哪个路由。

routerLink的值是一串包含了一组字符串数组（例如"['home']"）的字符串，看起来可能比较奇怪。这是因为在链接路由时，你可以提供更多信息。我们将在介绍子路由和路由参数时进行更加详尽的讲解。

我们暂时只使用来自于根应用组件的路由名字。

7.5　整合

现在有了所有的基本部件，可以来整合它们，实现路由导航了。

我们需要修改的第一个文件是应用程序的index.html。

下面是该文件的完整代码。

code/routes/basic/app/index.html

```
<!doctype html>
<html>
  <head>
    <base href="/">
    <title>ng-book 2: Angular Router</title>

    {% for (var css in o.htmlWebpackPlugin.files.css) { %}
      <link href="{%=o.htmlWebpackPlugin.files.css[css] %}" rel="stylesheet">
    {% } %}
  </head>
  <body>
    <router-app></router-app>
    <script src="/core.js"></script>
    <script src="/vendor.js"></script>
    <script src="/bundle.js"></script>
  </body>
</html>
```



描述htmlWebpackPlugin的部分来自于webpack模块捆绑器①。我们在本章中使用了webpack，它是一个帮你捆绑资源的工具。

你可能很熟悉这些代码，但是下面这行除外：

```
<base href="/">
```

① https://webpack.github.io/

这行声明了HTML标签base。传统上，该标签的作用是使用相对路径来告知浏览器去哪里查找图片和其他资源。

Angular的路由器也依赖这个标签来确定如何构建它的路由信息。

例如，如果一个路由的路径为/hello，base元素声明是href="/app"，那么应用程序将使用/app/#/hello作为实际路径。

有时候，Angular应用开发者对应用中HTML的head部分没有访问权。比如在重用已有大型应用的页头和页脚时。

幸运的是，我们有方法处理这种情况。你可以在配置NgModule时，像这样使用APP_BASE_HREF提供者，用代码来声明应用程序的基准路径：

```
@NgModule({
  declarations: [ RoutesDemoApp ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy },
    { provide: APP_BASE_HREF, useValue: '/' } // <--- this right here
  ]
})
```

将{ provide: APP_BASE_HREF, useValue: '/' }放到providers中，等同于在应用的HTML页头里使用<base href="/">。

7.5.1　创建组件

在处理主应用组件之前，首先创建三个简单的组件，每种路由各一个。

1. HomeComponent

HomeComponent只有一个h1标签，显示Welcome!。下面是HomeComponent的完整代码。

code/routes/basic/app/ts/components/HomeComponent.ts

```
/*
 * Angular
 */
import {Component} from '@angular/core';

@Component({
  selector: 'home',
  template: `<h1>Welcome!</h1>`
})
export class HomeComponent {
}
```

2. AboutComponent

同样，AboutComponent也只有一个基本的h1。

code/routes/basic/app/ts/components/AboutComponent.ts

```
/*
 * Angular
 */
import {Component} from '@angular/core';

@Component({
  selector: 'about',
  template: `<h1>About</h1>`
})
export class AboutComponent {
}
```

3. ContactComponent

AboutComponent也是一样。

code/routes/basic/app/ts/components/ContactComponent.ts

```
/*
 * Angular
 */
import {Component} from '@angular/core';

@Component({
  selector: 'contact',
  template: `<h1>Contact Us</h1>`
})
export class ContactComponent {
}
```

这些组件并没有什么特别之处，所以让我们开始探讨主app.ts文件。

7.5.2 应用程序组件

现在我们需要创建一个根级“应用程序”组件，将所有的部件组装起来。

我们先从core和router库导入需要的模块。

code/routes/basic/app/ts/app.ts

```
/*
 * Angular Imports
 */
import {
  NgModule,
  Component
} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
```

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {
  RouterModule,
  Routes
} from '@angular/router';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
```

接下来，导入上面创建的三个组件。

code/routes/basic/app/ts/app.ts

```
import {HomeComponent} from 'components/HomeComponent';
import {AboutComponent} from 'components/AboutComponent';
import {ContactComponent} from 'components/ContactComponent';
```

现在，让我们真正深入到组件代码之中。首先声明组件选择器和模板。

code/routes/basic/app/ts/app.ts

```
@Component({
  selector: 'router-app',
  template: `
 <div>
   <nav>
     <a>Navigation:</a>
     <ul>
       <li><a [routerLink]="['home']">Home</a></li>
       <li><a [routerLink]="['about']">About</a></li>
       <li><a [routerLink]="['contact']">Contact Us</a></li>
     </ul>
   </nav>

    <router-outlet></router-outlet>
  </div>
  `
})
class RoutesDemoApp {
}
```

我们将为这个组件使用两个路由指令：RouterOutlet和RouterLink。这两个指令和其他公共路由指令一起，在我们将RouterModule放置到NgModule的imports数组中时被导入进来。

作为回顾，RouterOutlet指令指定了路由内容在模板中被渲染的位置，即模板代码中<router-outlet></router-outlet>的位置。

RouterLink指令创建指向路由的导航链接。

code/routes/basic/app/ts/app.ts

```
      <a>Navigation:</a>
      <ul>
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['about']">About</a></li>
        <li><a [routerLink]="['contact']">Contact Us</a></li>
      </ul>
```

使用[routerLink]将指示Angular获取click事件的所有权，然后基于路由的定义，初始化路由器并导航到正确的位置。

7.5.3 配置路由

接下来，我们创建一组类型为Routes的对象数组，并用它来声明路由配置。

code/routes/basic/app/ts/app.ts

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },
];
```

在app.ts文件的最后，我们这样引导应用。

code/routes/basic/app/ts/app.ts

```
@NgModule({
  declarations: [
    RoutesDemoApp,
    HomeComponent,
    AboutComponent,
    ContactComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes) // <-- routes
  ],
  bootstrap: [ RoutesDemoApp ],
  providers: [
    { provide: LocationStrategy, useClass: HashLocationStrategy }
 ]
})
class RoutesDemoAppModule {}

platformBrowserDynamic().bootstrapModule(RoutesDemoAppModule)
  .catch((err: any) => console.error(err));
```

与一贯的做法一样，我们引导应用并指定RoutesDemoApp为根组件。

注意，我们将所有必需的组件放到declarations里。如果要路由到一个组件，那么必须在某个NgModule（当前模块或者导入的模块）里面声明它。

在imports中，我们有RouterModule.forRoot(routes)。RouterModule.forRoot(routes)是一个函数，接收我们的路由对象数组并配置路由器，然后返回依赖列表，例如RouteRegistry、Location和其他一些路由器运行时必需的类。

在providers中，我们有：

```
{ provide: LocationStrategy, useClass: HashLocationStrategy }
```

下面深入讲解这行代码的作用。

7.6　路由策略

定位策略（location strategy）是Angular应用从路由定义进行解析和创建路径的方式。



在AngularJS中，它被称作routing mode。

Angular的默认策略为`PathLocationStrategy`，也就是HTML5路由。在使用这个策略时，路由的路径是常规路径，例如/home或者/contact。

通过将`LocationStrategy`类绑定到新的策略类实例，我们可以改变应用的定位策略。

我们可以不使用默认的`PathLocationStrategy`，而是使用`HashLocationStrategy`。

我们使用锚点标记策略作为默认策略，因为如果使用HTML5路由，那么URL将成为普通的路径（而非使用锚点标记或者锚标签）。

这样，当你在客户端点击一个链接时，路由应该能正常工作并进行导航，比如从/about到/contact。

如果刷新页面，我们向服务器索要的就不是服务器提供的根URL，而是/about或者/contact。因为服务器端没有对应`/about`的页面，所以它会返回404.

该默认策略适用于基于锚点标记的路径，例如/#/home或者/#/contact。服务器将它们解析为/路径（这也是AngularJS的默认模式）。



如何在产品中使用HTML5模式呢？

要使用HTML5模式路由，你必须配置服务器来将所有“不存在”的路由重定向到根URL。

在routes/basic项目中，我们包含了一个脚本，可在webpack-dev-server环境下开发，并使用HTML5路径。

要使用它，需要cd routes/basic并运行node html5-dev-server.js。

最后，为了让示例应用适合这个新的策略，必须首先导入`LocationStrategy`和`HashLocationStrategy`。

code/routes/basic/app/ts/app.ts

```
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
```

然后将定位策略添加到`NgModule`的`providers`。

code/routes/basic/app/ts/app.ts

```
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```



你可以编写自己的策略。只需要扩展LocationStrategy类并实现一些方法即可。开始的好方法是阅读Angular的HashLocationStrategy或者PathLocationStrategy类的源代码。

7.7 路径定位策略

在示例应用的目录中，有一个名为app/ts/app.html5.ts的文件。

如果你想试试默认的PathLocationStrategy，那么将这个文件的内容复制到app/ts/app.ts中，然后重新加载应用即可。

7.8 运行应用程序

现在，你可以到应用的根目录（code/routes）并运行npm run server来启动应用程序。

当你在浏览器中输入http://localhost:8080/时，应该能看到home路由被渲染了（如图7-1所示）。

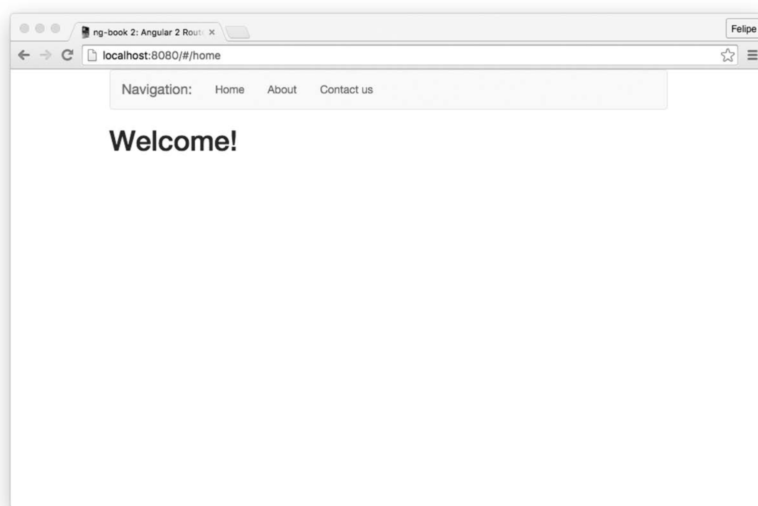


图7-1 Home路由

注意，浏览器中的URL被重定向到了http://localhost:8080/#/home。

现在点击每个链接，就会渲染相应的路由（分别如图7-2、图7-3所示）。

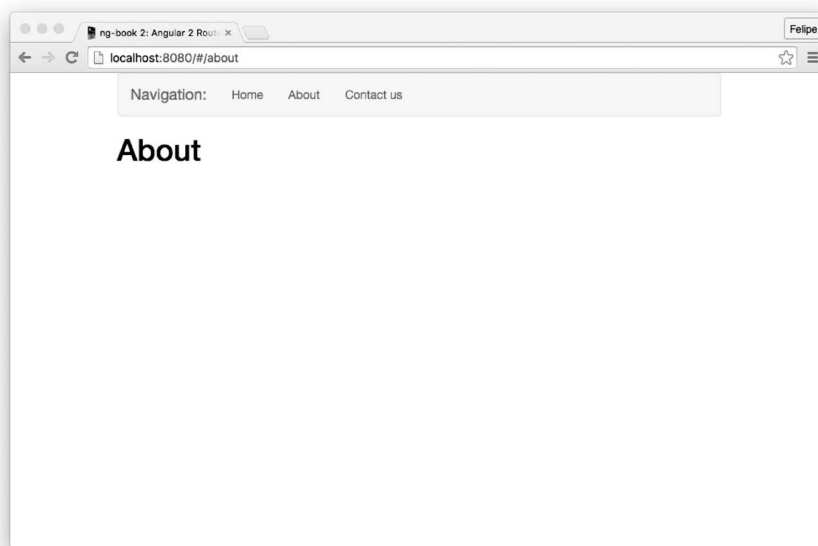


图7-2 About路由

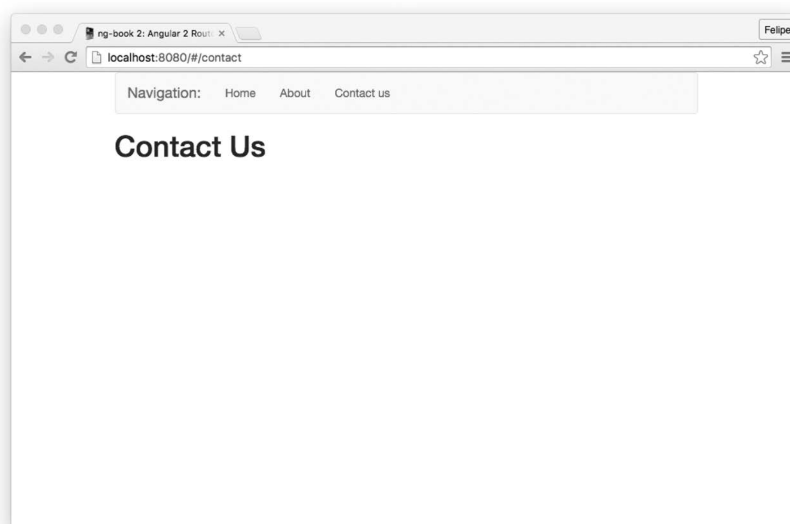


图7-3 Contact Us路由

7.9 路由参数

我们经常希望在应用程序中导航到特定的资源。例如，假设我们有一个新闻网站，它拥有很多文章。每篇文章可能有一个ID。如果有一篇ID为3的文章，那么可以通过下面的URL来导航到这篇文章：

```
/articles/3
```

如果有一篇ID为4的文章，我们可以在这里访问它：

```
/articles/4
```

以此类推。

很显然，我们不是为每篇文章编写一个路由，而是使用一个变量或者*路由参数*。我们可以像这样在路径段前面添加一个冒号，设定路由接收一个参数：

```
/route/:param
```

在示例新闻站里，我们可以这样定义路由：

```
/articles/:id
```

为了添加参数到路由配置，我们这样指定路由路径。

code/routes/music/app/ts/app.ts

```
const routes: Routes = [
  { path: '', redirectTo: 'search', pathMatch: 'full' },
  { path: 'search', component: SearchComponent },
  { path: 'artists/:id', component: ArtistComponent },
  { path: 'tracks/:id', component: TrackComponent },
  { path: 'albums/:id', component: AlbumComponent },
];
```

当我们访问路由`/artist/123`时，`123`部分是被传到路由的`id`路由参数。

但是，如何获取特定路由的参数呢？这正是使用路由参数的地方。

ActivatedRoute

为了使用路由参数，我们首先需要导入`ActivatedRoute`：

```
import { ActivatedRoute } from '@angular/router';
```

接下来，将`ActivatedRoute`注入组件的构造函数中。例如，假设我们有一个这样定义的`Routes`：

```
const routes: Routes = [
  { path: 'articles/:id', component: ArticlesComponent }
];
```

然后，在开发ArticleComponent时，我们将ActivatedRoute作为参数添加到构造函数：

```
export class ArticleComponent {
  id: string;

  constructor(private route: ActivatedRoute) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
}
```

注意，route.params是一个可观察对象。我们可以使用.subscribe将参数值提取到固定值。在这种情况下，我们将params['id']赋值给组件实例的变量id。

现在，在访问/articles/230时，组件的id属性应该接收230。

7.10 音乐搜索应用

下面来编写一个更加复杂的应用。我们将构建一个音乐搜索应用（如图7-4所示），它具有以下特性：

(1) 按照提供的关键词搜索曲目；

(2) 在数据表格中显示匹配曲目；

(3) 点击歌手名字时，显示歌手介绍；

(4) 点击专辑名字时，显示专辑信息和曲目列表；

(5) 点击歌曲名字时，显示曲目信息并允许用户试听。

这个应用需要的路由如下所示。

- /search：搜索表格和搜索结果。
- /artists/:id：艺术家信息，接收Spotify的ID为参数。
- /albums/:id：专辑信息，包含曲目列表，接收Spotify的ID。
- /tracks/:id：曲目信息和试听，也接收Spotify的ID。



示例代码 本节例子的完整代码可以在示例代码中的routes/music目录中找到。查阅README.md文件，了解构建和运行本例的步骤。

我们将使用Spotify API[①]来获取曲目、艺术家和专辑的信息。

① https://developer.spotify.com/web-api

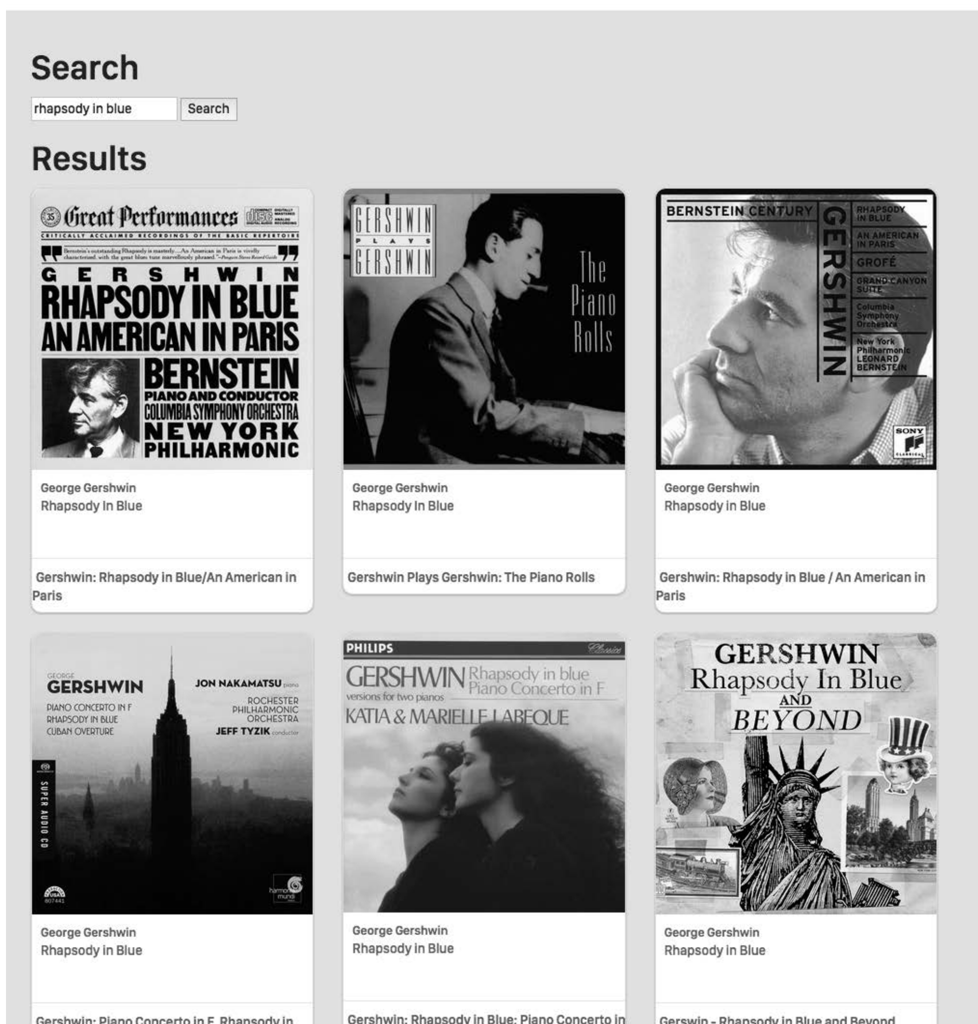


图7-4　音乐应用的搜索视图

7.10.1　首要步骤

我们要写的第一个文件是app.ts。首先，从Angular导入需要的类。

code/routes/music/app/ts/app.ts

```
/*
 * Angular Imports
```

```
 */
import {
  Component
} from '@angular/core';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import {
  RouterModule,
  Routes
} from '@angular/router';
import {
  LocationStrategy,
  HashLocationStrategy,
  APP_BASE_HREF
} from '@angular/common';

/*
 * Components
 */
```

现在我们有了所有导入声明，接下来考虑每个路由的组件。

- Search路由：新建SearchComponent。该组件将连接Spotify API并执行搜索功能，然后在数据表格中显示搜索结果。
- Artists路由：新建ArtistComponent，显示艺术家信息。
- Albums路由：新建AlbumComponent，显示专辑的曲目列表。
- Tracks路由：新建TrackComponent，显示曲目并允许试听。

因为新组件需要与Spotify API交互，所以我们需要创建一个服务，它使用http模块来调用API服务器。

应用的一切都依赖这些数据，所以我们首先创建SpotifyService。

7.10.2 SpotifyService



你可以在示例代码中的routes/music/app/ts/services目录找到SpotifyService的完整代码。

我们要实现的第一个方法是searchByTrack，它将利用提供的关键词来搜索曲目。

Spotify API文档中描述了API端点中有一个名为Search endpoint[①]的端点。

① https://developer.spotify.com/web-api/search-item/

该端点正是我们想要的：它接收一个查询对象（使用q参数）和一个type参数。

在这种情况下，查询对象是搜索关键词。因为搜索的是歌曲，所以type为track。

服务的第一个版本可能如下所示：

```
class SpotifyService {
  constructor(public http: Http) {
  }

  searchByTrack(query: string) {
    let params: string = [
      `q=${query}`,
      `type=track`
    ].join("&");
    let queryURL: string = `https://api.spotify.com/v1/search?${params}`;
    return this.http.request(queryURL).map(res => res.json());
  }
}
```

这段代码向https://api.spotify.com/v1/search这一URL执行HTTP GET请求，传入query（搜索关键词）和硬编码为track的type。

该http调用返回一个Observable。我们将进一步使用RxJS函数map转换搜索结果（一个http模块的Response对象）并将它解析为JSON，最终获得一个对象。

任何调用searchByQuery的函数都可以使用Observable API来订阅它的响应：

```
service
  .searchTrack('query')
  .subscribe((res: any) => console.log('Got object', res))
```

7.10.3　SearchComponent

现在我们有了执行曲目搜索的服务，可以开始编写SearchComponent了。

同样，以导入声明开始。

code/routes/music/app/ts/components/SearchComponent.ts

```
/*
 * Angular
 */

import {Component, OnInit} from '@angular/core';
import {
  Router,
  ActivatedRoute,
} from '@angular/router';

/*
 * Services
```

```
 */
import {SpotifyService} from 'services/SpotifyService';
```

这里，我们导入了刚刚新建的SpotifyService类和一些其他类。

我们的目标是像卡片一样一条一条地渲染曲目搜索结果，如图7-5所示。

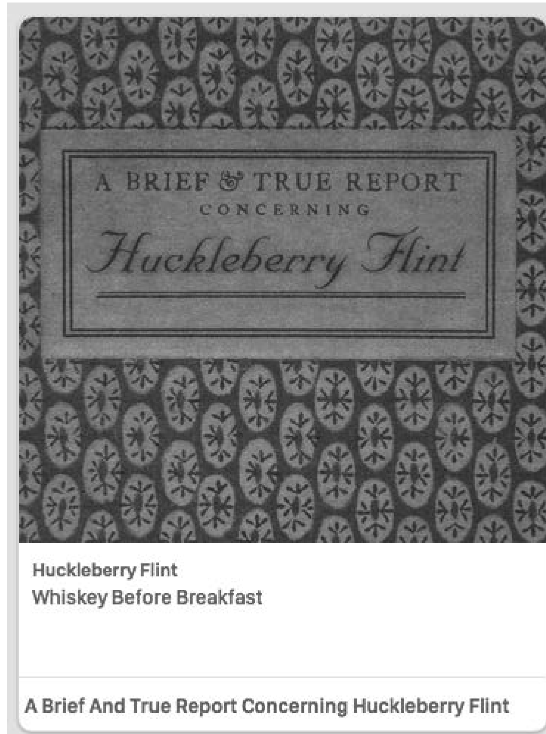


图7-5　音乐应用的卡片

现在开始开发组件。我们用search作为选择器，并使用下面的模板。该模板有点长，因为我们适当添加了一些样式，但是相比我们迄今做过的那些，它并不复杂。

code/routes/music/app/ts/components/SearchComponent.ts

```
@Component({
  selector: 'search',
  template: `
  <h1>Search</h1>

  <p>
    <input type="text" #newquery
      [value]="query"
      (keydown.enter)="submit(newquery.value)">
    <button (click)="submit(newquery.value)">Search</button>
  </p>
```

```
  <div *ngIf="results">
    <div *ngIf="!results.length">
      No tracks were found with the term '{{ query }}'
    </div>

    <div *ngIf="results.length">
      <h1>Results</h1>

      <div class="row">
        <div class="col-sm-6 col-md-4" *ngFor="let t of results">
          <div class="thumbnail">
            <div class="content">
              <img src="{{ t.album.images[0].url }}" class="img-responsive">
              <div class="caption">
                <h3>
                  <a [routerLink]="['/artists', t.artists[0].id]">
                    {{ t.artists[0].name }}
                 </a>
               </h3>
               <br>
               <p>
                  <a [routerLink]="['/tracks', t.id]">
                    {{ t.name }}
                  </a>
                </p>
              </div>
              <div class="attribution">
                <h4>
                  <a [routerLink]="['/albums', t.album.id]">
                    {{ t.album.name }}
                  </a>
                </h4>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
  `
})
```

7

1. 搜索框

下面来分段分析HTML模板。

搜索框在第一段中。

code/routes/music/app/ts/components/SearchComponent.ts

```
  <p>
    <input type="text" #newquery
      [value]="query"
```

```
    (keydown.enter)="submit(newquery.value)">
  <button (click)="submit(newquery.value)">Search</button>
</p>
```

这里，我们插入了输入框，并将其DOM元素的value属性绑定到组件的query属性。

我们还给这个元素赋予了一个模板变量，名为#newquery。这样我们就可以在模板中通过newquery.value来直接访问该输入框的值。

按钮将触发组件的submit方法，将输入框的值当作参数传入。

我们还希望在用户按下回车键以后触发submit事件，所以将keydown.enter事件绑定到输入框。

2. 搜索结果和链接

接下来的部分显示搜索结果。我们依靠ngFor指令来迭代返回对象中的每条曲目。

code/routes/music/app/ts/components/SearchComponent.ts

```
<div class="row">
  <div class="col-sm-6 col-md-4" *ngFor="let t of results">
    <div class="thumbnail">
```

我们为每条曲目显示其艺术家的名字。

code/routes/music/app/ts/components/SearchComponent.ts

```
<h3>
  <a [routerLink]="['/artists', t.artists[0].id]">
    {{ t.artists[0].name }}
  </a>
</h3>
```

注意我们是如何使用RouterLink指令来重定向到['/artists', t.artists[0].id]的。

这是为特定路由设置路由参数的方法。假设有一个id为abc123的艺术家，当这个链接被点击时，本应用将导航到/artist/abc123（abc123是:id参数）。

下面将展示如何在该路由对应的组件中获取这个参数。

现在，我们这样显示曲目：

code/routes/music/app/ts/components/SearchComponent.ts

```
<p>
  <a [routerLink]="['/tracks', t.id]">
    {{ t.name }}
  </a>
</p>
```

这样显示专辑：

code/routes/music/app/ts/components/SearchComponent.ts

```
            <h4>
              <a [routerLink]="['/albums', t.album.id]">
                {{ t.album.name }}
              </a>
            </h4>
```

3. SearchComponent类

先看看它的构造函数。

code/routes/music/app/ts/components/SearchComponent.ts

```
export class SearchComponent implements OnInit {
  query: string;
  results: Object;

  constructor(private spotify: SpotifyService,
              private router: Router,
              private route: ActivatedRoute) {
    this.route
      .queryParams
      .subscribe(params => { this.query = params['query'] || ''; });
  }
```

我们声明了两个属性：

- query，用来处理当前搜索关键词；
- results，用来存储搜索结果。

在构造函数的参数中，我们注入了（之前创建的）SpotifyService、Router和Activated-Route，并将它们设置为类属性。

在构造函数中，我们用subscribe订阅到queryParams属性。通过它，我们可以访问查询参数，比如搜索关键词（params['query']）。

在一个像http://localhost/#/search?query=cats&order=ascending这样的URL中，queryParams以对象的形式为我们提供路由参数。这就是说，我们可以从params['order']中访问order（在本例中为ascending）。

另外，注意queryParams与route.params有所不同。route.params在路由配置中匹配参数，而queryParams在查询字符串中匹配参数。

在本例中，如果没有query参数，那么我们将this.query设置为空字符串。

- **search方法**

在SearchComponent中，我们将调用SpotifyService服务并渲染搜索结果。我们要在下面两种情况下运行搜索：

- 当用户输入搜索关键词并提交表单时；
- 当用户使用带有查询参数的URL导航到本页面时（例如，用其他人共享的链接或者收藏的本页面链接）。

为了在上面两种情况下执行实际的搜索，我们创建了`search`方法。

code/routes/music/app/ts/components/SearchComponent.ts

```
search(): void {
  console.log('this.query', this.query);
  if (!this.query) {
    return;
  }

  this.spotify
    .searchTrack(this.query)
    .subscribe((res: any) => this.renderResults(res));
}
```

`search`函数通过当前`this.query`属性的值来得知应该搜索什么。因为我们在构造函数中订阅了`queryParams`，所以可以确认`this.query`总是有最新的搜索关键词。

然后，我们订阅到`searchTrack`可观察对象。这样，只要有新搜索结果到达，我们就调用`renderResults`。

code/routes/music/app/ts/components/SearchComponent.ts

```
renderResults(res: any): void {
  this.results = null;
  if (res && res.tracks && res.tracks.items) {
    this.results = res.tracks.items;
  }
}
```

我们声明了组件属性`results`。只要它的值有变化，Angular就会自动更新视图。

- *在页面加载时进行搜索*

正如上面指出的，我们希望在URL包含搜索查询参数时，能够直接自动获取搜索结果。

为了达到这个目标，我们将实现一个Angular路由器提供的钩子，在组件初始化的时候运行它。



这难道不是构造函数要做的吗？既正确，也不正确。正确是因为构造函数是用来初始化变量值的，但是如果想要撰写优质、容易测试的代码，你就要最小化对象**构建**的副作用。请记住，你应该像下面这样，将组件初始化代码放到一个钩子函数里。

下面是`ngOnInit`方法的代码。

code/routes/music/app/ts/components/SearchComponent.ts

```
  ngOnInit(): void {
    this.search();
  }
```

为了使用ngOnInit，我们导入OnInit接口，并声明组件类implements OnInit。

正如你所看到的，我们在这里仅仅执行了搜索。因为我们的搜索关键词来自于URL，所以这没有问题。

● 提交表单

现在来看看在用户提交表单的时候，我们应该干什么。

code/routes/music/app/ts/components/SearchComponent.ts

```
  submit(query: string): void {
    this.router.navigate(['search'], { queryParams: { query: query } })
      .then(_ => this.search() );
  }
```

我们手动告诉路由器导航到搜索路由，并提供了query参数，然后执行搜索功能。

这样做为我们带来了很大的好处：如果刷新浏览器，我们将会看到一样的搜索结果。可以说，我们将搜索关键词保存到URL了。

● 整合

下面是SearchComponent类的完整代码。

code/routes/music/app/ts/components/SearchComponent.ts

```
/*
 * Angular
 */

import {Component, OnInit} from '@angular/core';
import {
  Router,
  ActivatedRoute,
} from '@angular/router';

/*
 * Services
 */
import {SpotifyService} from 'services/SpotifyService';

@Component({
  selector: 'search',
  template: `
  <h1>Search</h1>

  <p>
```

```
      <input type="text" #newquery
        [value]="query"
        (keydown.enter)="submit(newquery.value)">
      <button (click)="submit(newquery.value)">Search</button>
    </p>

    <div *ngIf="results">
      <div *ngIf="!results.length">
        No tracks were found with the term '{{ query }}'
      </div>

      <div *ngIf="results.length">
        <h1>Results</h1>

        <div class="row">
          <div class="col-sm-6 col-md-4" *ngFor="let t of results">
            <div class="thumbnail">
              <div class="content">
                <img src="{{ t.album.images[0].url }}" class="img-responsive">
                <div class="caption">
                  <h3>
                    <a [routerLink]="['/artists', t.artists[0].id]">
                      {{ t.artists[0].name }}
                    </a>
                  </h3>
                  <br>
                  <p>
                    <a [routerLink]="['/tracks', t.id]">
                      {{ t.name }}
                    </a>
                  </p>
                </div>
                <div class="attribution">
                  <h4>
                    <a [routerLink]="['/albums', t.album.id]">
                      {{ t.album.name }}
                    </a>
                  </h4>
                </div>
              </div>
            </div>
          </div>
        </div>
     </div>
    </div>
    `
  })
  export class SearchComponent implements OnInit {
    query: string;
    results: Object;

    constructor(private spotify: SpotifyService,
                private router: Router,
                private route: ActivatedRoute) {
```

```
    this.route
      .queryParams
      .subscribe(params => { this.query = params['query'] || ''; });
  }

  ngOnInit(): void {
    this.search();
  }

  submit(query: string): void {
    this.router.navigate(['search'], { queryParams: { query: query } })
      .then(_ => this.search() );
  }

  search(): void {
    console.log('this.query', this.query);
    if (!this.query) {
      return;
    }

    this.spotify
      .searchTrack(this.query)
      .subscribe((res: any) => this.renderResults(res));
  }

  renderResults(res: any): void {
    this.results = null;
    if (res && res.tracks && res.tracks.items) {
      this.results = res.tracks.items;
    }
  }
}
```

7.10.4 尝试搜索

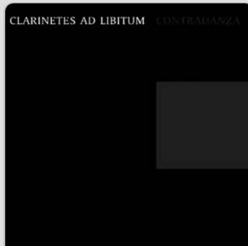
我们已经完成了搜索代码，现在来试一试（如图7-6所示）。











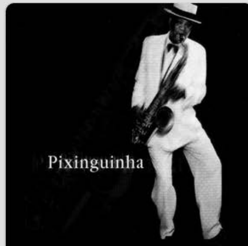


图7-6　尝试搜索

可以点击艺术家、曲目或者专辑链接来导航到相应的路由。

7.10.5　`TrackComponent`

我们用`TrackComponent`来处理曲目路由。它显示曲目名字和专辑封面图片，并允许用户使用HTML5的`audio`标签来进行试听。

code/routes/music/app/ts/components/TrackComponent.ts

```
template: `
<div *ngIf="track">
  <h1>{{ track.name }}</h1>
```

```
    <p>
      <img src="{{ track.album.images[1].url }}">
    </p>

    <p>
      <audio controls src="{{ track.preview_url }}"></audio>
    </p>

    <p><a href (click)="back()">Back</a></p>
  </div>
  `
```

和我们为搜索功能所做的一样，在这里使用Spotify API。让我们重构`searchTrack`方法，从中提取两个有用的方法，以供复用。

code/routes/music/app/ts/services/SpotifyService.ts

```
export class SpotifyService {
  static BASE_URL: string = 'https://api.spotify.com/v1';

  constructor(private http: Http) {
  }

  query(URL: string, params?: Array<string>): Observable<any[]> {
    let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;
    if (params) {
      queryURL = `${queryURL}?${params.join('&')}`;
    }

    return this.http.request(queryURL).map((res: any) => res.json());
  }

  search(query: string, type: string): Observable<any[]> {
    return this.query(`/search`, [
      `q=${query}`,
      `type=${type}`
    ]);
  }
```

现在，我们已经将这些方法分离到`SpotifyService`。注意`searchTrack`方法变得简单多了。

code/routes/music/app/ts/services/SpotifyService.ts

```
  searchTrack(query: string): Observable<any[]> {
    return this.search(query, 'track');
  }
```

然后创建一个方法，让我们正在开发的组件可以根据曲目的id来获取曲目信息。

code/routes/music/app/ts/services/SpotifyService.ts

```
  getTrack(id: string): Observable<any[]> {
    return this.query(`/tracks/${id}`);
  }
```

最后，在TrackComponent的新ngOnInit方法中调用getTrack。

code/routes/music/app/ts/components/TrackComponent.ts

```
ngOnInit(): void {
  this.spotify
    .getTrack(this.id)
    .subscribe((res: any) => this.renderTrack(res));
}
```

其他组件的工作原理很相似，它们都使用SpotifyService中的get*方法来根据id获取艺术家或曲目信息。

7.10.6　音乐搜索应用小结

现在，我们有了一个比较实用的音乐搜索和预览应用（如图7-7所示）。你可以试用它并搜索一些喜欢的音乐！



图7-7　完成路由之后

7.11　路由器钩子

在变换路由前，我们可能想要触发一些行为。典型的例子是用户认证。假设我们有登录路由和被保护的路由。

我们希望只有在登录页面中提供了正确的用户名和密码的时候，才允许应用导航到被保护的路由。

为了实现这个功能，我们需要连接到路由的生命周期钩子，并在激活被保护的路由时获得通知。然后调用一个认证服务，查询用户是否提供了正确的凭证。

要检查一个组件是否可以被激活，我们添加了一个守卫类到路由器配置的canActivate数组。

让我们再次修改最初的应用程序，添加用户名和密码输入框以及一个新的被保护的路由，该路由只在提供了指定的用户名和密码组合后才能被访问。



示例代码　本节例子的完整代码可以在示例代码中的routes/auth目录中找到。查阅README.md文件，了解构建和运行本例的步骤。

7.11.1 AuthService

我们来创建一个十分简单的最小化服务，负责认证和授权资源。

code/routes/auth/app/ts/services/AuthService.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class AuthService {
  login(user: string, password: string): boolean {
    if (user === 'user' && password === 'password') {
      localStorage.setItem('username', user);
      return true;
    }

    return false;
  }
```

login方法将在提供的用户名和密码为'user'和'password'时返回true。此外，在它们匹配时，使用localStorage来保存用户名。它标志着应用程序是否有一个仍然活跃的已登录用户。



如果你不熟悉，这里解释一下：localStorage是HTML5提供的键值对，用来在浏览器中保存信息。

它的API非常简单，仅仅包含了设置、读取和删除里面项目的方法。

参见MDN上的Storage文档[①]查看详情。

logout方法清除了username值。

code/routes/auth/app/ts/services/AuthService.ts

```
  logout(): any {
    localStorage.removeItem('username');
  }
```

① https://developer.mozilla.org/en-US/docs/Web/API/Storage

最后两个方法是：

- getUser，返回用户名或者null；
- isLoggedIn，使用getUser()并在有用户时返回true。

下面是这些方法的代码。

code/routes/auth/app/ts/services/AuthService.ts

```
getUser(): any {
  return localStorage.getItem('username');
}

isLoggedIn(): boolean {
  return this.getUser() !== null;
}
```

最后一件要做的事是导出一个AUTH_PROVIDERS，这样可以将其注入到应用中。

code/routes/auth/app/ts/services/AuthService.ts

```
export var AUTH_PROVIDERS: Array<any> = [
  { provide: AuthService, useClass: AuthService }
];
```

至此，我们有了用于注入到组件的AuthService服务，可以实现用户登录、检查当前登录用户和用户登出等。

随后，我们还将在路由器中使用它来保护ProtectedComponent。不过我们首先创建用于登录的组件。

7.11.2　LoginComponent

这个组件将在没有登录用户的时候显示登录表单，或者显示一条包含了用户信息和登录链接的小横幅。

下面是login和logout方法的代码。

code/routes/auth/app/ts/components/LoginComponent.ts

```
export class LoginComponent {
  message: string;

  constructor(private authService: AuthService) {
    this.message = '';
  }

  login(username: string, password: string): boolean {
    this.message = '';
    if (!this.authService.login(username, password)) {
      this.message = 'Incorrect credentials.';
      setTimeout(function() {
```

```
      this.message = '';
    }.bind(this), 2500);
  }
  return false;
}

logout(): boolean {
  this.authService.logout();
  return false;
}
```

在服务验证用户凭证后，我们就登入用户。

根据用户的登录状态，组件模板中有两段代码片段分别被显示出来。

第一段是登录表单，受到*ngIf="!authService.getUser()"保护。

code/routes/auth/app/ts/components/LoginComponent.ts

```
<form class="form-inline" *ngIf="!authService.getUser()">
  <div class="form-group">
    <label for="username">User:</label>
    <input class="form-control" name="username" #username>
  </div>

  <div class="form-group">
    <label for="password">Password:</label>
    <input class="form-control" type="password" name="password" #password>
  </div>

  <a class="btn btn-default" (click)="login(username.value, password.value)">
    Submit
  </a>
</form>
```

第二段是信息横幅，包含了登出链接，受到相反的*ngIf="authService.getUser()"保护。

code/routes/auth/app/ts/components/LoginComponent.ts

```
<div class="well" *ngIf="authService.getUser()">
  Logged in as <b>{{ authService.getUser() }}</b>
  <a href (click)="logout()">Log out</a>
</div>
```

另外，在出现验证错误时，会显示一段代码片段。

code/routes/auth/app/ts/components/LoginComponent.ts

```
<div class="alert alert-danger" role="alert" *ngIf="message">
  {{ message }}
</div>
```

现在我们就可以处理用户登录了，接下来创建想要被用户登录保护的资源。

7.11.3 ProtectedComponent 组件和路由守卫

1. ProtectedComponent

要保护组件，必先有组件。ProtectedComponent组件很简明。

code/routes/auth/app/ts/components/ProtectedComponent.ts

```
/*
 * Angular
 */
import {Component} from '@angular/core';

@Component({
  selector: 'protected',
  template: `<h1>Protected content</h1>`
})
export class ProtectedComponent {
}
```

我们希望只有登录的用户可以访问这个组件。但是如何才能做到呢?

答案是使用路由器钩子canActivate，连接到一个实现CanActivate接口的守卫类。

2. LoggedInGuard守卫

新建一个名为guards的目录，然后新建loggedIn.guard.ts文件。

code/routes/auth/app/ts/guards/loggedIn.guard.ts

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { AuthService } from 'services/AuthService';

@Injectable()
export class LoggedInGuard implements CanActivate {
  constructor(private authService: AuthService) {}

  canActivate(): boolean {
    return this.authService.isLoggedIn();
  }
}
```

该守卫声明了它实现CanActivate接口。可以通过实现canActivate方法满足这个声明。

我们注入AuthService到这个类的构造函数，并将其保存到私有变量authService。

在canActivate函数中，我们通过this.AuthService来检查用户的登录状态isLoggedIn。

3. 配置路由器

为了使用这个守卫，我们需要这样配置路由器：

(1) 导入LoggedInGuard;

(2) 在路由配置中使用`LoggedInGuard`；

(3) 添加`LoggedInGuard`到提供者列表中（这样它就可以被注入了）。

我们在app.ts中实现以上步骤。

首先导入`LoggedInGuard`。

code/routes/auth/app/ts/app.ts

```
import {AUTH_PROVIDERS} from 'services/AuthService';
import {LoggedInGuard} from 'guards/loggedIn.guard';
```

然后将带有守卫的`canActivate`添加到被保护的路由。

code/routes/auth/app/ts/app.ts

```
const routes: Routes = [
  { path: '',          redirectTo: 'home', pathMatch: 'full' },
  { path: 'home',      component: HomeComponent },
  { path: 'about',     component: AboutComponent },
  { path: 'contact',   component: ContactComponent },
  { path: 'protected', component: ProtectedComponent,
    canActivate: [LoggedInGuard]}
];
```

最后将`LoggedInGuard`添加到提供者列表中。

code/routes/auth/app/ts/app.ts

```
  providers: [
    AUTH_PROVIDERS,
    LoggedInGuard,
    { provide: LocationStrategy, useClass: HashLocationStrategy },
  ]
```

4. 用户登录

我们必须添加：

code/routes/auth/app/ts/app.ts

```
import {LoginComponent} from 'components/LoginComponent';
```

然后添加：

(1) 一个新链接，指向被保护的路由；

(2) `<login>`标签到模板中，用来渲染新组件。

下面是app.ts的代码。

code/routes/auth/app/ts/app.ts

```
@Component({
  selector: 'router-app',
  template: `
```

7

```
  <div class="page-header">
    <div class="container">
      <h1>Router Sample</h1>
      <div class="navLinks">
        <a [routerLink]="['/home']">Home</a>
        <a [routerLink]="['/about']">About</a>
        <a [routerLink]="['/contact']">Contact Us</a>
        <a [routerLink]="['/protected']">Protected</a>
      </div>
    </div>
  </div>

  <div id="content">
    <div class="container">

      <login></login>

      <hr>

      <router-outlet></router-outlet>
    </div>
  </div>
  `
})
class RoutesDemoApp {
  constructor(private router: Router) {
  }
}
```

现在，在浏览器打开应用时，我们可以看到新的登录表单和被保护的链接（如图7-8所示）。

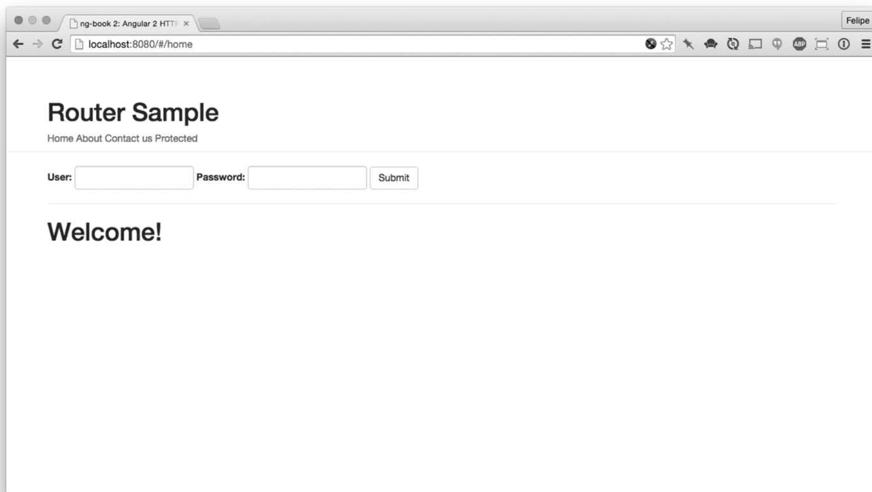


图7-8　认证应用：初始页

如果点击被保护的链接，什么也不会发生。手动访问http://localhost:8080/#/protected的效果也是一样。

在表单中输入用户名和密码，点击Submit按钮。你将看到一条显示了当前用户的横幅（如图7-9所示）。

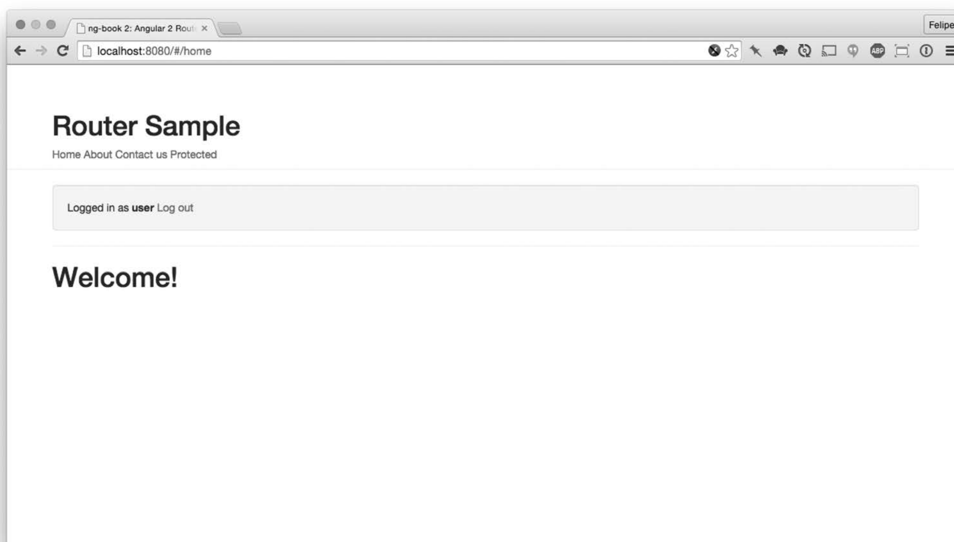


图7-9　认证应用：登录后

正如我们所料，在点击被保护的链接时，我们被重定向了，而且组件也被渲染了（如图7-10所示）。



安全注意事项：在过于依赖客户端路由保护为我们提供安全性之前，理解它的工作机制是至关重要的。实际上，你应该把客户端路由保护看作**用户体验**的一种形式，而不是安全的一种形式。

归根到底，应用的所有JavaScript代码都会服务于客户端。不管用户是否已经登录，这些代码都能被检测到。

因此，如果有敏感数据需要保护，你必须使用**服务器端认证**来保护它们。也就是说，对每条查询数据的请求，都要求用户提供一个服务器验证的有效API密钥（或者认证令牌）。

构建完整的认证系统超出了本书的范围。最重要的是要明白，在客户端保护路由并不一定会阻挡任何人查看这些路由背后的JavaScript页面。

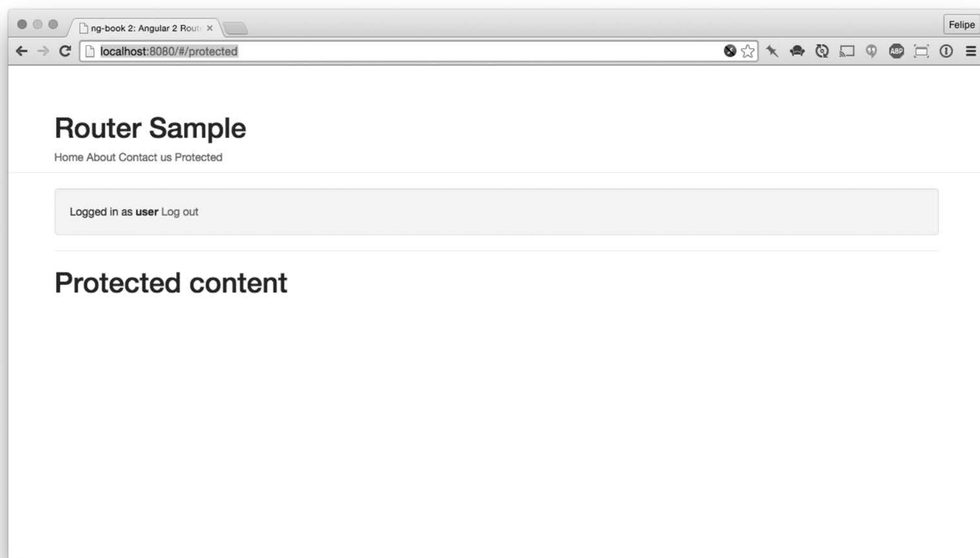


图7-10　认证应用：受保护区域

7.12　嵌套路由

嵌套路由是在一些路由中包含其他路由。利用嵌套路由，我们可以封装父级路由的功能，并在它的子级路由中使用这些功能。

假设我们有个网站，它有一个“我们是谁？”区域，允许用户了解我们的团队。它还有一个叫作“产品”的区域。

我们可能认为“我们是谁？”的完美路由是/about，“产品”的完美路由是/products。

然后，在访问这些区域时，我们很高兴地显示了所有团队和所有产品。

但是，如果随着网站的成长，我们需要显示团队中每个人的个人信息以及每种产品的信息该怎么办？

为了支持这种情况，路由器要允许用户定义嵌套路由。

你可以有多重嵌套的`router-outlet`。这样，应用的每个区域都可以有自己的子组件，这些组件也可以有自己的`router-outlet`。

下面用一个示例进行讲解。

在本例中，我们有一个产品区，用户在其中可以通过访问一个特殊的URL查看两种推荐的产

品。对于其他的产品，路由会使用产品ID。



示例代码　本节例子的完整代码可以在示例代码中的routes/nested目录中找到。查阅README.md文件，了解构建和运行本例的步骤。

7.12.1　配置路由

首先在app.ts文件中描述两种顶级路由。

code/routes/nested/app/ts/app.ts

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'products', component: ProductsComponent, children: childRoutes }
];
```

home路由看起来很眼熟；注意products有个children参数。它是从哪儿来的？我们在定义ProductsComponent时定义了childRoutes。

7.12.2　ProductsComponent 组件

这个组件有自己的路由配置。

code/routes/nested/app/ts/components/ProductsComponent.ts

```
export const routes: Routes = [
  { path: '', redirectTo: 'main',pathMatch: 'full'},
  { path: 'main', component: MainComponent },
  { path: ':id', component: ByIdComponent },
  { path: 'interest', component: InterestComponent },
  { path: 'sportify', component: SportifyComponent },
];
```

注意，在第一个对象上面有个空的path。这么做是为了在访问/products时重定向到main路由。

我们要看的另一个路由是:id。在这种情况下，当用户访问一些没有可以匹配的路由时，此路由就会垫底。在/之后传进来的一切都将被提取为路由的参数，即id。

然后在组件的路由器中为每种静态子路由添加一个链接。

code/routes/nested/app/ts/components/ProductsComponent.ts

```
    <a [routerLink]="['./main']">Main</a> |
    <a [routerLink]="['./interest']">Interest</a> |
    <a [routerLink]="['./sportify']">Sportify</a> |
```

可以看到路由链接的格式都是['./main']，前面有./。它表明了导航到main路由是相对于

当前路由上下文的。

你也可以用['products', 'main']的形式声明路由。这么做的坏处是，子路由知晓父路由；如果想要移动或者复用该组件，可能需要重新编写路由链接。

添加链接后，我们添加一个输入框让用户可以输入产品ID，以及一个按钮在点击后导航到该产品。最后添加了router-outlet。

code/routes/nested/app/ts/components/ProductsComponent.ts

```
  template: `
  <h2>Products</h2>

  <div class="navLinks">
    <a [routerLink]="['./main']">Main</a> |
    <a [routerLink]="['./interest']">Interest</a> |
    <a [routerLink]="['./sportify']">Sportify</a> |
    Enter id: <input #id size="6">
    <button (click)="goToProduct(id.value)">Go</button>
  </div>

  <div class="products-area">
    <router-outlet></router-outlet>
  </div>
  `
```

让我们看看ProductsComponent的代码。

code/routes/nested/app/ts/components/ProductsComponent.ts

```
export class ProductsComponent {
  constructor(private router: Router, private route: ActivatedRoute) {
  }

  goToProduct(id:string): void {
    this.router.navigate(['./', id], {relativeTo: this.route});
  }
}
```

首先，我们在构造函数中声明了一个Router的实例变量，因为我们将使用该实例来通过id导航到产品。

想要查看某产品时，我们使用goToProduct方法。在goToProduct方法中，我们调用路由器的navigate方法并提供路由名字和包含路由参数的对象。在本例中，我们简单地传递了id。

注意，我们在navigate函数中使用相对路径./。为了使用相对路径，我们还要将一个relativeTo对象作为选项传入，它告诉路由器究竟是相对于哪个路由。

运行应用程序，我们将看到主页，如图7-11所示。

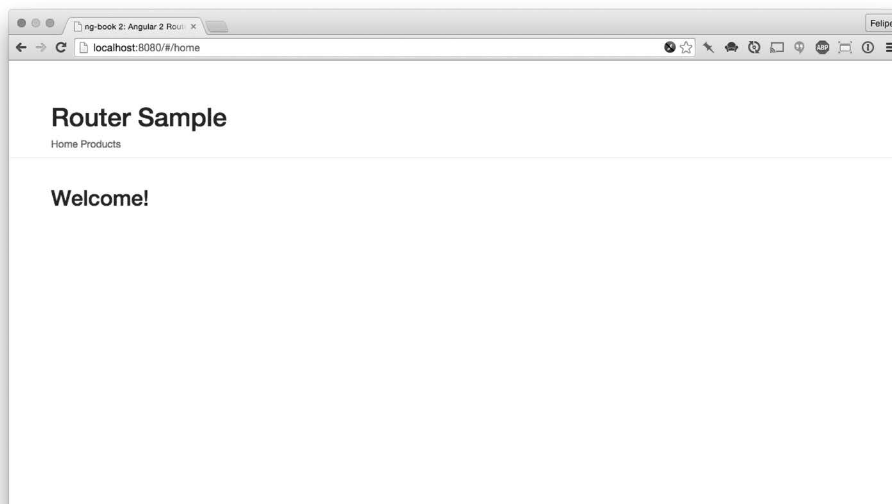


图7-11　嵌套的路由应用

如果点击产品链接，你将被重定向到/products/main，如图7-12所示。

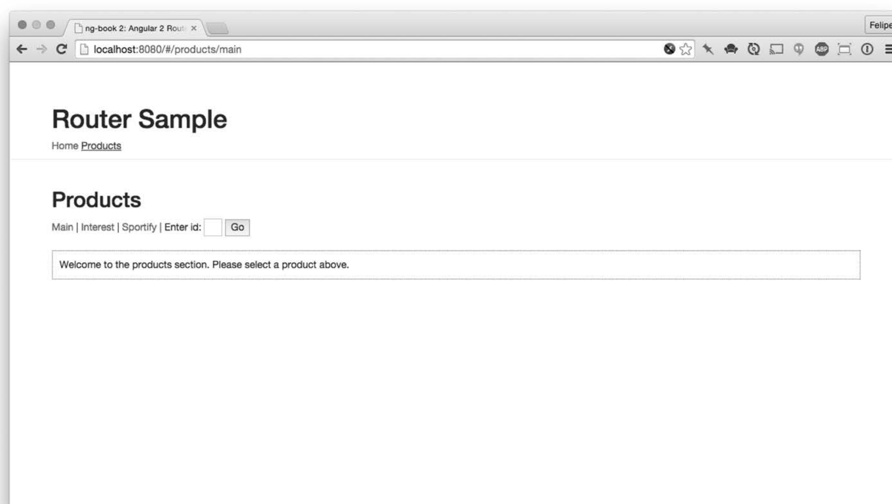


图7-12　嵌套的路由应用：产品区

灰色细线下面的所有内容都是使用主应用的`router-outlet`来渲染的。

虚线方框里面的内容是在`ProductComponent`的`router-outlet`中渲染的。这就是配置父级和子级路由分别进行渲染的方法。

当访问其中一个产品链接时，或者在文本框中输入id并点击Go按钮后，新的内容将在ProductComponent组件中的路由出口中渲染（如图7-13所示）。

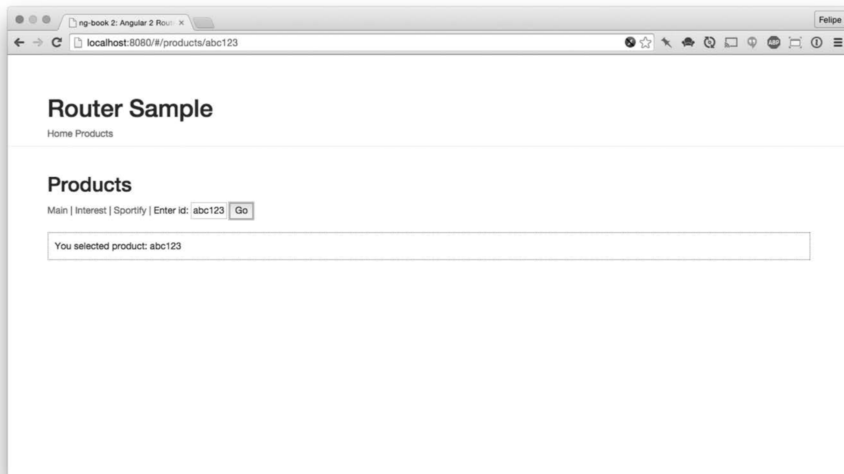


图7-13　嵌套路由应用：按ID查询产品

另外，值得注意的是Angular的路由器很智能，它会优先使用具体路由（比如/products/spotify），然后才使用参数化的路由（比如/products/123）。这样，/products/sportify将不会被更加通用的、捕捉所有路由的/products/:id处理。

嵌套路由的重定向和链接

作为回顾，我们使用['myRoute']来导航到名为MyRoute的顶级路由。但是，只有当你在同样的顶级上下文中时，这种方法才可行。

在子级组件中，如果你试图链接或重定向到['myRoute']，路由器将试图寻找一个兄弟路由，故而出错。在这种情况下，使用以斜杠开头的['/myRoute']。

同样，在顶级上下文中，如果想要链接或重定向到一个子级路由，我们需要使用路由定义数组的多个元素。

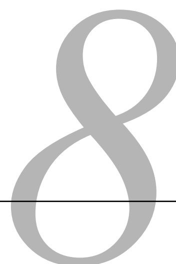
假设我们想要访问Show路由；它是Product路由的子级。在这种情况下，我们使用['product', 'show']，正如路由定义所示。

7.13　总结

正如我们所看到的，全新的Angular路由器非常强大和灵活。现在就在你的应用中使用路由器吧！

第 8 章

依赖注入



随着程序规模的增长，我们常常遇到应用模块需要相互通信的情况。当模块A需要模块B才能运行时，我们就说B是A的依赖。

获取依赖的最常见方式之一就是直接导入（import）一个文件。例如，在某个假想模块中，我们可以这么做：

```
// in A.ts
import {B} from 'B'; // a dependency!

B.foo(); // using B
```

通常，只要导入其他代码就足够了；但是在某些情况下，要用到更加精巧的方式提供依赖。

- 如果我们想在测试时把B的实现替换为MockB，该怎么办呢？
- 如果我们想在整个应用中共享B类的单一实例（比如单例模式），该怎么办呢？
- 如果我们想在每次用到B类时都创建一个新实例（比如工厂模式），该怎么办呢？

依赖注入可以解决这些问题。

依赖注入（dependency injection，DI）是这样一个系统：它让程序中的某部分可以访问其他部分，而且我们可以配置它们的访问方式。



可以把注入器看作new操作符的替代品。

依赖注入这个术语既被用来描述一种设计模式（可用于很多种框架），也被用来指代Angular内置的DI实现库。

使用依赖注入技术的主要优点是客户代码不必知晓如何创建依赖，它们只需要与那些依赖交互就可以了。

8.1　注入示例：PriceService

假设我们有一个Product类。每个产品都有一个基准价格。我们要靠一个服务来计算该产品的含税价，它需要如下输入：

- 产品的基准价格
- 销售时所在的州[①]

下面是不使用依赖注入时的代码：

```
class Product {
  constructor(basePrice: number) {
    this.service = new PriceService();
    this.basePrice = basePrice;
  }

  price(state: string) {
    return this.service.calculate(this.basePrice, state);
  }
}
```

想象一下，我们要为此Product类写一个测试。假设这个PriceService类要使用数据库查询来获得产品在指定州的税率。如果这样写测试的话：

```
let product;

beforeEach(() => {
  product = new Product(11);
});

describe('price', () => {
  it('is calculated based on the basePrice and the state', () => {
    expect(product.price('FL')).toBe(11.66);
  });
})
```

尽管这个测试可以工作，但是暴露了一些缺陷。为了让这个测试成功运行，需要满足两个前提条件：

(1) 数据库必须保持运行；

(2) 佛罗里达州（代号FL）的税率必须始终像我们期望的一样。

根本原因在于：Product类和PriceService类（而它又依赖于数据库）之间突兀的强烈依赖会让我们的测试变得更脆弱。

如果稍微改写一下Product类呢？

---

① 在美国，不同州的税率有所不同。——译者注

```
class Product {
  constructor(service: PriceService, basePrice: number) {
    this.service = service;
    this.basePrice = basePrice;
  }

  price(state: string) {
    return this.service.calculate(this.basePrice, state);
  }
}
```

现在，当要创建Product的实例时，客户方代码可以决定把PriceService的哪个具体实现传给这个新实例了。

这样，只要创建一个mock版本的PriceService类就可以大幅简化测试了：

```
class MockPriceService {
  calculate(basePrice: number, state: string) {
    if (state === 'FL') {
      return basePrice * 1.06;
    }

    return basePrice;
  }
}
```


基于这个小改动，我们就可以微调测试，移除它对数据库的依赖：

```
let product;

beforeEach(() => {
  const service = new MockPriceService();
  product = new Product(service, 11);
});

describe('price', () => {
  it('is calculated based on the basePrice and the state', () => {
    expect(product.price('FL')).toBe(11.66);
  });
})
```

另一个好处是我们现在能更加确信自己正在不受外界影响地测试Product类。也就是说，我们能确保该类正在使用一个行为上可预测的依赖。

8.2 “别打给我们……”

这种注入依赖的技术是基于一项被称为控制反转的设计原则。



控制反转（inversion of control，IoC）原则的非正式称谓是“好莱坞法则”。它来自好莱坞的一句常用语“别打给我们，我们会打给你（don’t call us, we’ll call you）”。

多年以来，它在与全应用语境相关的部件（指组件、服务、管道等Angular代码块）中用得非常普遍，也常被用来解决依赖的创建和设置问题。这一点在例子中体现得很清楚：`Product`类不得不了解`PriceService`类。

问题在于，一旦部件变得过于关心它的依赖，部件本身就会变得脆弱而难以修改。如果我们修改了一个部件，这项修改就会向上扩散到所有依赖它的部件中。它会影响到程序中很多不同的区域，甚至可能超出程序的边界。换句话说，这些部件之间产生了紧耦合。

使用依赖注入，我们就可以得到一个更加松耦合的架构。这时，当修改单一部件时，对程序中其他区域的影响就小多了。同时，只要这些部件之间的接口没有变，我们甚至可以在不修改其他部件中实现代码的情况下集体更换它们。

Angular从AngularJS中继承来的一项伟大特性就是它们都使用这种控制反转模式。Angular使用自带的依赖注入机制来解析这些依赖。

在传统方式下，如果部件A需要依赖部件B，那就意味着A要在内部创建一个B的实例，也就是A依赖于B（如图8-1所示）。

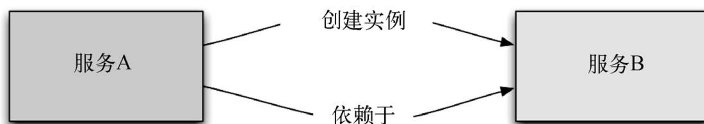


图8-1　不用依赖注入框架时

Angular利用依赖注入机制改变了这一点。在这种机制下，如果需要在部件A中用到部件B，我们就应该期待B被传给A（如图8-2所示）。

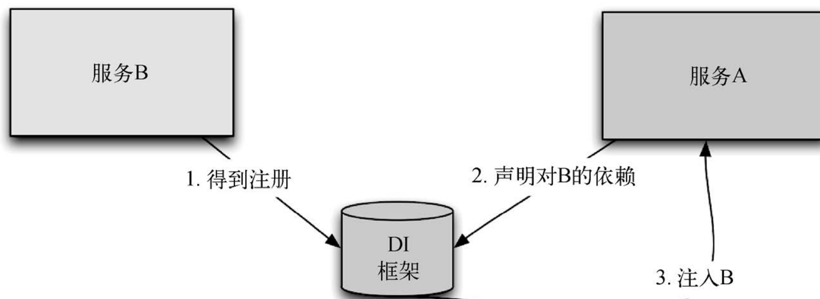


图8-2　使用依赖注入框架时

在传统场景下，这带来了很多优点。一个优点就是：如果我们准备单独测试A，可以创建一个mock版本的B，并把它注入到A中。

在本书的前面，我们已经多次用过服务和依赖注入了，比如在第7章创建音乐应用时。为了与Spotify API交互，我们创建了SpotifyService。它被注入到了很多部件中，比如下面这个来自AlbumComponent的片段。

code/routes/music/app/ts/components/AlbumComponent.ts

```
export class AlbumComponent implements OnInit {
  id: string;
  album: Object;

  constructor(private route: ActivatedRoute,
              private spotify: SpotifyService, // <-- injected
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
```

现在，我们就来学习如何创建自己的服务以及能用哪些形式注入它们吧。

8.3 依赖注入的部件

要注册一个依赖，我们就得找到一些东西作为那个依赖的标识。这个标识被称为依赖的令牌（token）。比如，如果我们想要注册某个API的URL，就可以用字符串API_URL作为令牌。同样，如果我们要注册一个类，就可以使用这个类本身作为它的令牌，就像我们即将看到的。

在Angular中，依赖注入包括如下三部分。

- 提供者（也常被称为绑定）负责把一个令牌（可能是字符串也可能是类）映射到一个依赖的列表。它告诉Angular该如何根据指定的令牌创建对象。
- 注入器负责持有一组绑定；当外界要求创建对象时，解析这些依赖并注入它们。
- 依赖就是将被用于注入的对象。

我们可以借助图8-3来理解它们各自扮演的角色。

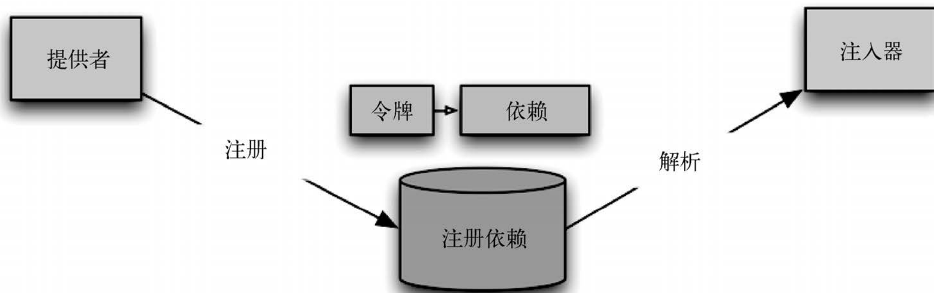


图8-3 依赖注入

与依赖注入打交道时，有很多不同的选项，我们来分别看看它们的用途。

最常见的情况是提供一个服务或值，它将在整个应用中保持一致。在我们的应用中，99%的场景可能都属于这种情况。

既然这就是我们要做的一切，那就在下一节示范怎样写一个基本的服务吧，因为它正是我们在开发大多数应用的大部分时间里所需要的。

说的够多了，开始编码！

8.4 尝试注入器

就像前面提到过的，Angular会在幕后帮我们设置好依赖注入。不过，在我们和注解打交道并且把依赖注入集成到部件中之前，自己先尝试使用一下注入器。

先来创建一个直接返回字符串的示例服务。

code/dependency_injection/injector/app/ts/app.ts

```
/*
 * The injectable service
 */
class MyService {
  getValue(): string {
    return 'a value';
  }
}
```

接下来，创建该应用的组件。

code/dependency_injection/injector/app/ts/app.ts

```
@Component({
  selector: 'di-sample-app',
  template: `
  <button (click)="invokeService()">Get Value</button>
  `
})
class DiSampleApp {
  myService: MyService;

  constructor() {
    let injector: any = ReflectiveInjector.resolveAndCreate([MyService]);
    this.myService = injector.get(MyService);
    console.log('Same instance?', this.myService === injector.get(MyService));
  }

  invokeService(): void {
    console.log('MyService returned', this.myService.getValue());
  }
}
```

下面对这个过程进行分解。我们首先声明了DiSampleApp组件，它会渲染出一个按钮。当点击此按钮时就会调用invokeService方法。

仔细看该组件的构造函数就会发现，我们正在使用一个来自ReflectiveInjector的静态方法，名为resolveAndCreate。该方法负责创建一个新的注入器。我们传给它的参数是一个数组，其中是这个新注入器需要知道的可供注入物。在这个例子中，它知道MyService这个可注入物就够了。



ReflectiveInjector是Injector的一个具体实现，它使用反射（reflection）机制来找出正确的参数类型。虽然也有一些别的注入器，不过在大多数应用中，ReflectiveInjector应该是最常用的“常规”注入器了。

需要注意的一点是：它会注入该类的一个单例对象。

这可以从构造函数中的最后两行得到验证。首先要求刚创建的注入器给我们一个MyService类的实例，然后把它存入组件的myService字段。之后，在console.log函数中要求注入器再次给我们一个MyService的实例，并输出它与myService字段进行比较的结果：

```
console.log('Same instance?', this.myService === injector.get(MyService));
```

我们可以在控制台中确认这两个实例确实是指向同一个对象的引用：

```
Same instance? true
```

注意，由于使用了自己的注入器，我们并不需要在启动时把MyService加入NgModule的providers列表中。

code/dependency_injection/injector/app/ts/app.ts

```
@NgModule({
  declarations: [ DiSampleApp ],
  imports: [ BrowserModule ],
  bootstrap: [ DiSampleApp ]
})
class DiSampleAppModule {}

platformBrowserDynamic().bootstrapModule(DiSampleAppModule);
```

8.5 用 NgModule 提供依赖

不过，在正常情况下，还是得告诉NgModule要注入哪些提供者。

比如，我们想让该MyService单例对象在整个应用中都能被注入。

为了能够注入，必须把它们添加到NgModule的providers属性中。示例代码如下：

```
@NgModule({
```

```
  declarations: [
    MyAppComponent,
    // other components ...
  ],
  providers: [ MyService ]   // <--- here
})
class MyAppModule {}
```

这样，MyAppComponent就能把MyService注入构造函数中了：

```
export class MyAppComponent {

  constructor(private myService: MyService /* <-- injected */) {
    // do something with myService here
  }

  // ...
}
```

当我们把这个类本身放进providers中时：

```
  providers: [ MyService ]
```

就是在告诉Angular：当MyService被注入时，我们希望提供MyService的一个单例实例。因为这种需求非常普遍，所以这个类实际上是一种缩写形式，其等价的完整配置方式是：

```
  providers: [
    { provide: MyComponent, useClass: MyComponent }
  ]
```

除了创建类的实例之外，还有很多其他的方式可以进行注入，接下来就来逐个查看。

8.6　提供者

Angular的依赖注入体系有很多精巧之处，其中之一是我们有很多种方式来配置注入过程。比如可以：

- 注入一个类的（单例）实例；
- 调用任意函数，并注入该函数的返回结果；
- 注入一个值；
- 创建一个别名。

下面分别用例子进行解释。

8.6.1　使用类

注入类的单例实例大概是最常见的注入类型了。

配置方法如下：

```
{ provide: MyComponent, useClass: MyComponent }
```

需要注意的是：provide配置方法接收两个键（key）。第一个provide键是我们用作这个可注入对象标识的令牌，第二个useClass键用来指出注入什么以及如何注入。

在这里，我们把MyComponent类映射到了MyComponent令牌。在这个例子中，类名和令牌名是匹配的。这是最常见的情况，但是必须知道：令牌和被注入物并不一定同名。

如前所见，该例子中的注入器将会在幕后创建一个单例对象，并在每次注入它时返回同一个实例。

当然，首次注入时它尚未实例化，需要创建一个MyComponent实例。此时，依赖注入系统就会调用该类的构造函数。

如果服务的构造函数需要一些参数，会怎么样呢？假设我们有这样一个服务。

code/dependency_injection/misc/app/ts/app.ts

```
class ParamService {
  constructor(private phrase: string) {
    console.log('ParamService is being created with phrase', phrase);
  }

  getValue(): string {
    return this.phrase;
  }
}
```


注意，它的构造函数需要传入一个短语作为参数。如果我们使用标准注入机制，就会在浏览器中看到一个错误，如图8-4所示。

```
Cannot resolve all parameters for 'ParameterService'(?). Make sure that all the          lang.js:375
parameters are decorated with Inject or have valid type annotations and that 'ParameterService' is
decorated with Injectable.
```

图8-4 注入错误

这是因为我们没有为注入器提供足够的信息来构造这个类。要解决这个问题，就得告诉注入器在创建该服务的实例时要使用哪个参数。

如果想在创建服务时传入一个参数，就要改用工厂了。

8.6.2 使用工厂

如果要使用工厂进行注入，就需要写一个返回任意对象的函数。

```
{
  provide: MyComponent,
  useFactory: () => {
    if (loggedIn) {
```

```
      return new MyLoggedComponent();
    }
    return new MyComponent();
  }
}
```

注意，在这个例子中，我们注入时用的令牌是MyComponent，但是它会检查（作用域外面的）loggedIn变量。如果loggedIn为真，则注入器会返回一个MyLoggedComponent的实例；否则返回MyComponent的实例。

工厂还可以拥有自己的依赖：

```
{
  provide: MyComponent,
  useFactory: (user) => {
    if (user.loggedIn()) {
      return new MyLoggedComponent(user);
    }
    return new MyComponent();
  },
  deps: [ User ]
}
```

因此，如果要使用前面的ParamService，我们就得把它用useFactory包裹起来。

code/dependency_injection/misc/app/ts/app.ts

```
@NgModule({
  declarations: [ DiSampleApp ],
  imports: [ BrowserModule ],
  bootstrap: [ DiSampleApp ],
  providers: [
    SimpleService,
    {
      provide: ParamService,
      useFactory: (): ParamService => new ParamService('YOLO')
    }
  ]
})
class DiSampleAppAppModule {}

platformBrowserDynamic().bootstrapModule(DiSampleAppAppModule)
  .catch((err: any) => console.error(err));
```



我们可以把SimpleService直接放在providers列表中，这是因为Simple-Service并不需要任何参数。它会被翻译成：

```
{ provide: SimpleService, useClass: SimpleService }
```

可以说，工厂是创建可注入对象的最强方式，因为我们可以在工厂函数中“为所欲为”。

8.6.3 使用值

当我们需要一个常量，而它可能会根据应用的其他部分甚至环境进行重定义时（比如测试环境或生产环境），这种方式非常有用。

```
{ provide: 'API_URL', useValue: 'http://my.api.com/v1' }
```

在8.9节中，我们会提供一个更完善的例子。

8.6.4 使用别名

我们还可以制造一个别名来引用以前注册过的令牌，比如：

```
{ provide: NewComponent, useClass: MyComponent }
```

8.7 应用中的依赖注入

当我们开发应用时，需要经过三步才能进行依赖注入：

(1) 创建该服务的类；

(2) 在准备接受注入的部件上声明该依赖；

(3) 配置要注入的依赖（比如在我们的`NgModule`中通过Angular注册要注入的依赖）。

我们要做的第一件事是创建该服务的类，该类会暴露出我们想要用到的那些行为。它也被称为*可注入对象*，因为它就是我们的部件将通过依赖注入接收到的东西。

下面示范如何创建服务。

code/dependency_injection/simple/app/ts/services/ApiService.ts

```
export class ApiService {
  get(): void {
    console.log('Getting resource...');
  }
}
```

现在已经有了要注入的东西，接下来要声明当Angular创建部件时，我们希望接收哪些依赖。

我们以前直接使用`Injector`类，但在写部件时，我们通常会使用Angular提供的两种快捷方式。

第一种是在部件的构造函数中声明这些可注入对象。这也是最典型的用法。

要做到这一点，必须先导入该服务。

code/dependency_injection/simple/app/ts/app.ts

```
/*
 * Services
```

8

```
 */
import { ApiService } from 'services/ApiService';
```

然后在构造函数中声明它。

code/dependency_injection/simple/app/ts/app.ts

```
class DiSampleApp {
  constructor(private apiService: ApiService) {
  }
```

当我们在组件的构造函数中声明依赖时，Angular会通过反射机制来找出要注入的类。也就是说，Angular会发现我们正在构造函数中查找一个`ApiService`类型的对象，并检查依赖注入系统以找出合适的可注入对象。

有时我们需要给Angular更多的提示，来告诉它我们到底要注入什么。在这种情况下，我们要使用第二种方式，即`@Inject`注解。

```
class DiSampleApp {
  private apiService: ApiService;
  constructor(@Inject(ApiService) apiService) {
    this.apiService = apiService;
  }
```



如果我们要用这种等价形式，可以打开app.long.ts文件，把它的内容复制到app.ts中。

使用依赖注入的最后一步是把部件想要的东西与可注入对象关联起来。换句话说，我们告诉Angular：当部件声明了它的依赖时，应该注入什么。

```
    { provide: ApiService, useClass: ApiService }
```

在这个例子中，我们使用令牌`ApiService`暴露出了`ApiService`类的单例对象。

最后，我们把这个`ApiService`添加到`NgModule`的`providers`属性中。

code/dependency_injection/simple/app/ts/app.ts

```
@NgModule({
  declarations: [ DiSampleApp ],
  imports: [ BrowserModule ],
  bootstrap: [ DiSampleApp ],
  providers: [ ApiService ] // <-- here
})
class DiSampleAppAppModule {}

platformBrowserDynamic().bootstrapModule(DiSampleAppAppModule)
  .catch((err: any) => console.error(err));
```

8.8 使用注入器

我们已经和注入器打过交道了，现在要更进一步，看看什么时候需要显式地使用它们。

情况之一是，当我们需要控制在什么时机创建依赖的单例对象时。

为了说明什么时候会出现这种情况，我们来构建另一个应用。除了使用我们以前创建过的ApiService外，它还会用到一个新的服务。

该服务将用来根据浏览器的窗口大小实例化另外两个服务。如果窗口宽度小于800像素，它就返回一个名叫SmallService的服务实例；否则返回LargeService的实例。

下面是SmallService的代码。

code/dependency_injection/complex/app/ts/services/SmallService.ts

```
export class SmallService {
  run(): void {
    console.log('Small service...');
  }
}
```

下面是LargeService的代码。

code/dependency_injection/complex/app/ts/services/LargeService.ts

```
export class LargeService {
  run(): void {
    console.log('Large service...');
  }
}
```

然后，我们开始写ViewPortService，它负责在两者之间作出选择。

code/dependency_injection/complex/app/ts/services/ViewPortService.ts

```
import {LargeService} from './LargeService';
import {SmallService} from './SmallService';

export class ViewPortService {
  determineService(): any {
    let w: number = Math.max(document.documentElement.clientWidth,
                             window.innerWidth || 0);

    if (w < 800) {
      return new SmallService();
    }
    return new LargeService();
  }
}
```

现在，我们创建一个使用这些服务的应用。

code/dependency_injection/complex/app/ts/app.ts

```
class DiSampleApp {
  constructor(private apiService: ApiService,
              @Inject('ApiServiceAlias') private aliasService: ApiService,
              @Inject('SizeService') private sizeService: any) {
  }
```

这里我们仍然用以前的方式获得一个ApiService的实例。不过这次我们通过别名'ApiServiceAlias'获得了同一个实例。最后，我们要获得一个'SizeService'的实例，但它还没有定义过。

为了理解每个服务都代表什么，我们来看看NgModule。

code/dependency_injection/complex/app/ts/app.ts

```
@NgModule({
  declarations: [ DiSampleApp ],
  imports: [ BrowserModule ],
  bootstrap: [ DiSampleApp ],
  providers: [
    ApiService,
    ViewPortService,
    { provide: 'ApiServiceAlias', useExisting: ApiService },
    {
      provide: 'SizeService',
      useFactory: (viewport: any) => {
        return viewport.determineService();
      },
      deps: [ViewPortService]
    }
  ]
})
class DiSampleAppAppModule {}
```

这段代码的意思是，我们首先希望该应用的注入器知道ApiService和ViewPortService这两个可注入对象。

接下来的声明表示是我们希望通过另一个令牌（字符串ApiServiceAlias）来使用既有服务ApiService。

然后，我们通过另一个字符串令牌SizeService定义了另一个可注入对象。该工厂通过把ViewPortService列在自己的deps数组中，表明自己需要接收该服务的一个实例。然后，它将调用该实例的determineService()方法，并根据浏览器的宽度返回一个SmallService或LargeService的实例。

当点击模板中的一个按钮时，我们会发起三次调用：一次是对ApiService，一次是对别名ApiServiceAlias，最后一次则是对SizeService。

code/dependency_injection/complex/app/ts/app.ts

```
  invokeApi(): void {
    this.apiService.get();
    this.aliasService.get();
    this.sizeService.run();
  }
```

现在，如果我们运行此应用并在小型浏览器窗口中点击Invoke API按钮，结果会如图8-5所示。

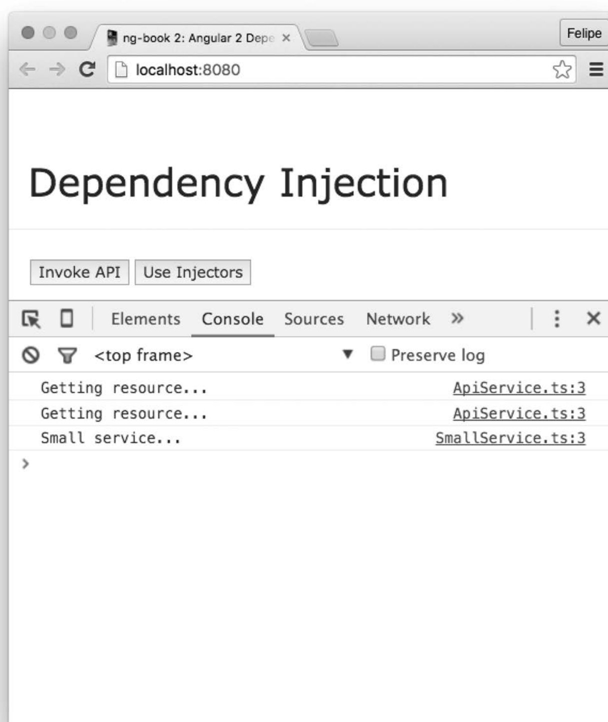


图8-5　小型浏览器窗口

我们会获得三条日志：一条来自ApiService，另一条来自别名服务，最后一条来自SmallService。

如果我们让浏览器窗口更大一点，刷新页面并再次点击按钮，结果会如图8-6所示。

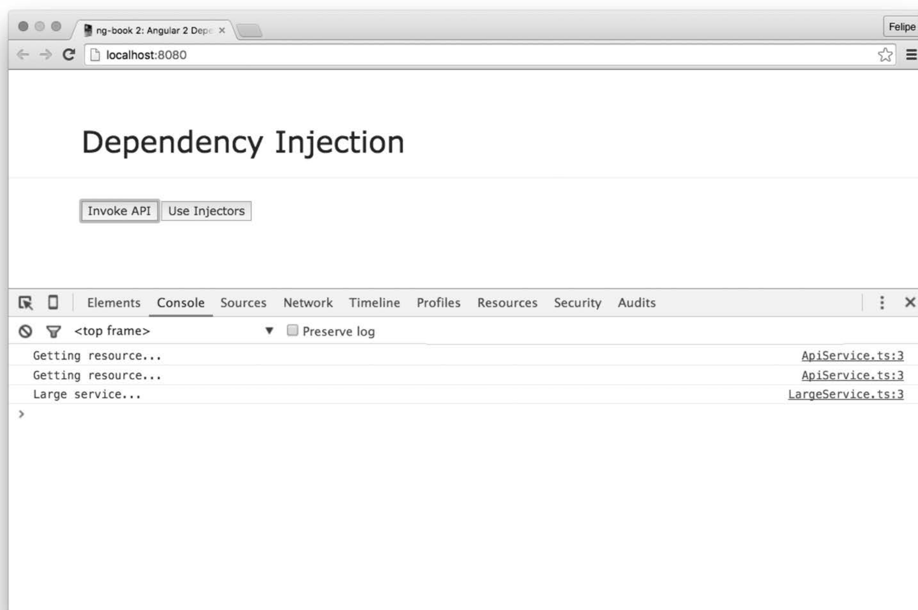


图8-6　大型浏览器窗口

这样我们就会收到来自LargeService的日志。然而，如果把浏览器窗口调小一点，不刷新页面并再次点击按钮，收到的仍将是来自LargeService的日志，如图8-7所示。

这是因为这个工厂函数只会被执行一次，也就是在应用启动时。

要解决这个问题，我们可以创建自己的注入器，并通过如下方式获得正确的服务实例。

code/dependency_injection/complex/app/ts/app.ts

```
useInjectors(): void {
  let injector: any = ReflectiveInjector.resolveAndCreate([
    ViewPortService,
    {
      provide: 'OtherSizeService',
      useFactory: (viewport: any) => {
        return viewport.determineService();
      },
      deps: [ViewPortService]
    }
  ]);
  let sizeService: any = injector.get('OtherSizeService');
  sizeService.run();
}
```

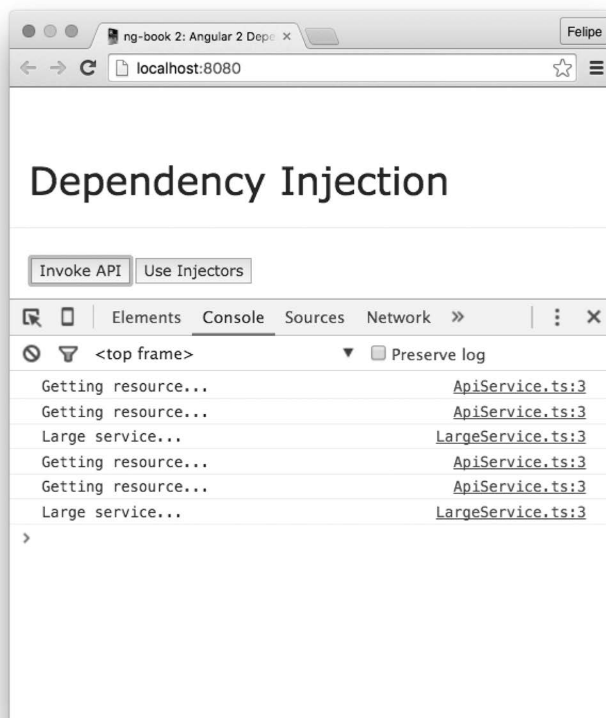


图8-7 小型浏览器窗口：调整大小后

这里我们创建了一个注入器，它知道ViewPortService和另一个以字符串OtherSizeService为令牌的可注入对象。这个可注入对象与我们以前用过的SizeService使用同一个工厂。

最后，它使用我们创建的注入器来获得一个OtherSizeService的实例。

这时，如果我们在一个大型浏览器窗口中运行该应用并点击Use Injector按钮，就会收到一条来自LargeService的日志。然而，如果我们把窗口调小，即使不刷新页面，也能正常收到来自SmallService的日志。这是因为现在注入器是随需创建的，我们每次点击按钮时都会重新执行工厂函数。这真漂亮！

8.9 替换值

使用依赖注入的另一个理由是在运行期间改变被注入对象的硬编码值。当我们用一个API服务来向应用的后端API发起HTTP请求时，就会出现这种情况。在单元测试或集成测试的场景下，我们肯定不希望代码接触生产环境的数据库。这时，就可以写一个Mock的API服务，它可以天衣

无缝地替换掉我们的具体实现。我们这就来详细解释一下。

比如，如果在开发环境下运行该应用，我们可能会接触与生产环境下不同的API服务器。

当我们发布一个开源或可复用的服务时，这就更加有用了。这种情况下，我们要允许调用者定义或改写API的URL。

我们来写一个简单的示例应用，它会根据自己是在生产模式还是开发模式运行来为API的URL注入不同的值。先从ApiService类开始。

code/dependency_injection/value/app/ts/services/ApiService.ts

```
import { Inject } from '@angular/core';

export const API_URL: string = 'API_URL';

export class ApiService {
  constructor(@Inject(API_URL) private apiUrl: string) {
  }

  get(): void {
    console.log(`Calling ${this.apiUrl}/endpoint...`);
  }
}
```

我们先声明了一个常量，它会被用作API URL依赖的令牌。换句话说，Angular会根据字符串'API_URL'来存储要调用哪个URL的信息。这样，当我们使用@Inject(API_URL)时，就会把正确的值注入到apiUrl变量中。

注意，我们还同时导出了API_URL常量，这样客户方应用就可以从服务之外使用API_URL来注入正确的值。

现在，我们已经有了服务，接下来写一个应用组件，它将使用该服务，并根据所在的运行环境为URL提供不同的值。

code/dependency_injection/value/app/ts/app.ts

```
@Component({
  selector: 'di-value-app',
  template: `
  <button (click)="invokeApi()">Invoke API</button>
  `
})
class DiValueApp {
  constructor(private apiService: ApiService) {
  }

  invokeApi(): void {
    this.apiService.get();
  }
}
```

这是组件的源代码。在构造函数中，我们声明了一个ApiService类型的变量apiService。这时，Angular就能推断出我们需要一个ApiService型的依赖，并在运行时注入它。如果我们要让它更明确一点，那么可以这样写：

```
constructor(@Inject(ApiService) private apiService: ApiService) {
}
```

该组件有一个Invoke API按钮。当点击此按钮时，我们调用ApiService的get()方法。此方法就会把我们正在使用的API_URL的值记录到控制台中。

下一步是使用提供者来配置本应用。

code/dependency_injection/value/app/ts/app.ts

```
const isProduction: boolean = false;

@NgModule({
  declarations: [ DiValueApp ],
  imports: [ BrowserModule ],
  bootstrap: [ DiValueApp ],
  providers: [
    { provide: ApiService, useClass: ApiService },
    {
      provide: API_URL,
      useValue: isProduction ?
        'https://production-api.sample.com' :
        'http://dev-api.sample.com'
    }
  ]
})
class DiValueAppAppModule {}

platformBrowserDynamic().bootstrapModule(DiValueAppAppModule)
```

我们首先声明了一个名叫isProduction的常量，并把它设置为false。我们先假装做了点什么来检测自己是否是在生产模式下运行。这里可以先把它硬编码进去，也可以使用一些小技巧来实现它，比如使用webpack和一个.env文件。

最后，我们引导本应用，并设置两个提供者：一个用真正的实现类来提供ApiService，另一个则用来提供API_URL。如果在生产模式下运行，我们就使用某个值，否则用另一个。

要测试它，我们可以带上isProduction = true来运行本应用。然后点击该按钮，就会看到日志中记录了生产模式下的URL，如图8-8所示。

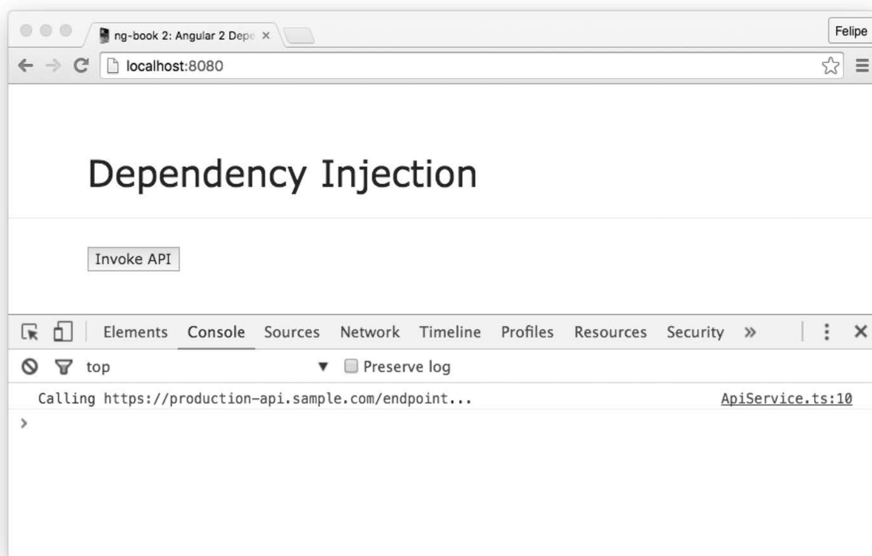


图8-8　生产环境

如果把它改成`isProduction = false`，就会看到开发模式下的URL，如图8-9所示。



图8-9　开发环境

8.10 NgModule

NgModule是帮助编译器和依赖注入对依赖进行组织的方式。让我们看看为什么需要NgModule以及它们是如何工作的。

我们要剖析的是Angular中的编译器和依赖注入这两个角色。简而言之，Angular需要解决组件定义了哪些HTML标记（tag）以及这些依赖来自哪里这两个问题。

8.10.1 NgModule 与 JavaScript 模块

你可能会疑惑：为什么我们需要一个新的模块系统呢？只用ES6/TypeScript的模块还不够吗？

这是因为虽然仍然要用import来把代码模块加载到JavaScript环境中，但NgModule体系却是Angular框架内部对依赖进行组织的一种方式。特别是围绕两个问题：编译出了哪些标记以及哪些依赖应该被注入其中。

8.10.2 编译器与组件

8

对于编译器来说，如果有一个带有自定义标记的Angular模板，你就得告诉编译器哪些标记是有效的（以及应该为它们附加上哪些功能）。

比如，假设你有这样一个组件：

```
@Component({
  selector: 'hello-world',
  template: `<div>Hello world</div>`
})
class HelloWorld {
}
```

我们希望编译器知道下列HTML代码应该使用这个hello-world组件（这个hello-world可不是随便写的无效标签）：

```
<div>
  <hello-world></hello-world>
</div>
```

在AngularJS中，hello-world选择器应该已经在全局范围注册过了。在你的应用成长到发生命名冲突之前，这样做都很方便。比如，如果两个开源项目使用了相同的选择器，问题就很难解决。

如果你用过Angular RC.5之前的老版本，可能还记得那些版本需要你在@Component注解中指定一个directives选项。这种方式的优点是它不怎么需要“魔术”来移除表面的冲突。它的问

题在于要为每个组件指定用到的所有指令，这样太繁琐了。

改用NgModule，我们可以在“模块”一级告诉Angular组件的依赖关系。我们会在稍后讲解更多内容。

8.10.3　依赖注入与提供者

回忆一下，依赖注入是一种让依赖在整个应用中可用的组织形式。它对简单的import代码形式进行了强化，让我们得以用一种标准化的方式来共享单例、创建工厂以及在测试期间改写依赖。

在Angular RC.5之前的版本中，我们不得不在bootstrap函数的providers参数中指定待注入的一切（提供者）。



回想下列术语：**提供者**提供（创建、实例化等）你想要的**可注入对象**。在Angular中，当你想要访问**可注入对象**时，就把一个依赖**注入**一个函数中。Angular中的依赖注入框架就会找到它，并把它提供给你。

现在，利用NgModule，每个提供者都被指定为模块的一部分。

现在你应该明白了为什么需要NgModule以及要怎样使用它了吧？这里是最简单的例子：

```
// app.ts

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ HelloWorld ],
  bootstrap: [ HelloWorld ]
})
class HelloWorldAppModule {}

platformBrowserDynamic().bootstrapModule(HelloWorldAppModule);
```

在这里，我们定义了一个HelloWorldAppModule类，随后将其作为我们应用程序的入口点。从RC5开始，不再使用组件来引导应用，而是改用bootstrapModule，就像这里的代码一样。

NgModule可以导入其他模块作为自己的依赖。我们要在浏览器中运行此应用，所以还要导入BrowserModule。

我们要在此应用中使用HelloWorld组件。记住这里的关键：每个组件都必须在某些NgModule中声明过。这里我们把HelloWorld放在了NgModule的declarations中。

我们说HelloWorld组件从属于HelloWorldAppModule；任何组件都只能从属于一个NgModule。

我们通常会把很多组件一起放进一个NgModule中，这很像Java中的package或C#中的namespace。

如果你想引导该模块（也就是把该模块作为应用的入口点），那么就得提供一个bootstrap属性，用它来指定一个作为该模块入口点的组件。

在这个例子中，你将会bootstrap这个HelloWorld组件，并把它作为根组件。不过，如果你创建的模块不需要用作应用程序入口点，那么bootstrap属性就是可选的。

8.10.4 组件可见性

要使用任何组件，当前的NgModule都必须先知道它。假设我们想在hello-world组件中使用user-greeting组件，就像这样：

```
<!-- hello-world template -->
<div>
  <user-greeting></user-greeting>
  world
</div>
```

如果任何组件想要使用其他组件，它必须首先通过NgModule体系获得访问权。有两种方式能做到这一点：

(1) user-greeting组件位于同一个NgModule中（比如HelloWorldAppModule）；

(2) HelloWorldAppModule导入（imports）了UserGreeting组件所在的模块。

假设我们要访问第二个路由。下面是UserGreetingModule中的UserGreeting组件的实现代码：

```
@Component({
  selector: 'user-greeting',
  template: `<span>hello</span>`
})
class UserGreeting {
}

@NgModule({
  declarations: [ UserGreeting ],
  exports: [ UserGreeting ]
})
export class UserGreetingModule {}
```

注意，这里我们添加了一个新的属性exports。可以先把exports当作这个NgModule中公开组件的列表。这里隐含的意思是，我们可以轻松地制作一个私有组件，只要别把它列进exports中就行了。

如果你忘了把组件加到declarations和exports中（然后还要在另一个模块中通过imports引入本模块），那么组件将不会生效。为了让你的组件能在其他模块中通过imports的方式使用，你必须把组件同时放在这两个地方。

现在，只要把它导入到HelloWorldAppModule中，我们就可以在HelloWorld组件中使用它了，就像这样：

```
// updated HelloWorldAppModule

@NgModule({
  declarations: [ HelloWorld ],
  imports: [ BrowserModule, UserGreetingModule ], // <-- added
  bootstrap: [ HelloWorld ],
})
class HelloWorldAppModule {}
```

8.10.5　指定提供者

只要把可注入对象的提供者添加到NgModule的providers属性中，就算完成指定了。

例如，假设我们有这样一个简单的服务：

```
export class ApiService {
  get(): void {
    console.log('Getting resource...');
  }
}
```

我们希望把它注入到组件中，就像这样：

```
class ApiDataComponent {
  constructor(private apiService: ApiService) {
  }

  getData(): void {
    this.apiService.get();
  }
}
```

用NgModule可以很容易做到这一点：只要把ApiService传给该模块的providers属性就可以了：

```
@NgModule({
  declarations: [ ApiDataComponent ],
  providers: [ ApiService ]   // <-- here
})
class ApiAppModule {}
```

这里直接传入ApiService实际上是一个缩写版本，使用provide的完整版本是这样的：

```
@NgModule({
  declarations: [ ApiDataComponent ],
  providers: [
    provide(ApiService, { useClass: ApiService })
  ]
})
class ApiAppModule {}
```

我们是在告诉Angular，当ApiService被注入时，依赖注入体系要负责创建、维护该类的单例，并把它传进去。

要从其他模块中使用这些提供者，必须先导入（import）那个模块。

由于ApiDataComponent和ApiService都位于同一个NgModule中，ApiDataComponent可以直接注入ApiService。否则，就需要把包含ApiService的模块导入到ApiAppModule中。

8.11　总结

可以看出，依赖注入和NgModule的协作为管理应用中的依赖提供了一种强大的方式。要了解更多知识，请参考下列资源：

- Angular DI官方文档[1]
- Victor Savkin对AngularJS中和Angular中依赖注入的对比[2]

8

① https://angular.io/docs/ts/latest/guide/dependency-injection.html
② http://victorsavkin.com/post/126514197956/dependency-injection-in-angular-1-and-angular-2

第 9 章

Angular数据架构

9

数据架构概览

管理数据可以说是编写可维护应用最棘手的方面之一。有很多种方法可以将数据应用到你的应用之中：

- AJAX HTTP请求
- Websocket
- Indexdb
- LocalStorage
- LocalStorage
- Service Worker
- 等等

数据架构涉及的问题如下。

- 如何将所有不同的数据源聚合成一个完整的体系？
- 如何防止意想不到的副作用导致bug？
- 如何更好地构建代码以使其更容易维护并让新来的团队成员更容易上手？
- 当数据发生变化时，如何让应用尽快作出反应？

多年以来，MVC一直是构建数据应用的标准模式：模型包含业务逻辑，视图负责显示数据，控制器将所有一切联系在一起。不过问题是，我们知道MVC模式并不能很好地直接转化到客户端的网络应用中。

目前，数据架构领域出现了复兴并有许多新理念涌现出来。

- MVW/双向数据绑定：Model-View-Whatever[1]是用来形容AngularJS中默认架构的一个术语。`$scope`提供数据双向绑定，整个应用都共用同样的数据结构，某个区域的一个变化

① https://plus.google.com/+AngularJS/posts/aZNVhj355G2

会传达至该应用的其余部分。

- Flux[1]：它使用单向数据流。在Flux中，Store负责存储数据，View负责渲染Store中的数据，Action负责改变Store中的数据。虽然设置Flux有一点繁琐，但是因为数据只在一个方向上流动，所以很容易推断。
- 可观察对象：observable给我们提供了数据流。我们订阅数据流然后执行操作对变化作出反应。RxJS[2]是当下最流行的响应式JavaScript库，给我们提供了强有力的操作符，用来在数据流上组合一系列操作。



还有很多关于这些理念的变种，例如：

Flux作为一种模式而并非具体实现，它有许多不同的实现方案（就像MVC有许多的实现方案一样）；

- Immutability是以上所有数据架构的一个常见变种；
- Falcor[3]是一个强大的框架，可以帮你将客户端模型和服务端数据进行绑定。
- Falcor通常使用可观察对象类型的数据架构。

Angular 数据架构

Angular在数据架构的选择上极其灵活。一种数据策略在一个项目中可行并不代表在另一个项目中也可行，所以Angular并未规定具体的技术栈，而是力图让你无论选择何种数据架构都能很容易使用（同时保持高性能）。

这样的好处是，你可以拥有足够的灵活性来让Angular适应几乎任何情况。只是有一点不太好：你将不得不自己选择适合项目的数据架构。

别担心，我们不会让你自己去作出这个艰难的决定！在接下来的几章里，我们将教你如何使用这里提到的某些模式来构建应用。

① https://facebook.github.io/flux/

② https://github.com/Reactive-Extensions/RxJS

③ http://netflix.github.io/falcor/

第 10 章

使用可观察对象的数据架构，第1部分：服务

10.1 可观察对象和 RxJS

在Angular中，可以使用可观察对象作为数据架构的骨架来构建应用。使用可观察对象构造数据被称为响应式编程（reactive programming）。

可观察对象和响应式编程究竟是什么呢？响应式编程是一种处理异步数据流的编程方法。可观察对象是用来实现响应式编程的主要数据结构。必须承认，这些术语可能不怎么明确。因此，我们会在本章通过具体的例子来帮助你更好地理解这些概念。

10.1.1 注意：一些必备的 RxJS 相关知识

需要指出的是，本书的重点不是讲解响应式编程。有一些其他不错的资源可以教会你响应式编程的基础，你应该阅读它们。我们在下面列举了几个。

你可以将本章视为如何使用RxJS和Angular的入门教程，而不是RxJS和响应式编程的详细指南。

本章会详细解释我们接触到的RxJS概念和API，但如果RxJS对你来说还是个新鲜事物，那么你可能需要通过其他相关资源来补充知识。



本章使用Underscore.js

Underscore.js[①]是一个流行的类库，为`Array`和`Object`这样的JavaScript数据结构提供函数式操作符。本章将在使用RxJS的同时大量使用它。如果在代码中看见了`_`，比如`_.map`或者`_.sortBy`，要知道这就是在使用Underscore.js类库。要查阅Underscore.js文档，请阅读http://underscorejs.org/。

① http://underscorejs.org/

10.1.2 学习响应式编程和 RxJS

如果你只想学习RxJS，推荐阅读这篇文章。

- Andre Staltz的“你不容错过的响应式编程入门”（https://gist.github.com/staltz/868e7e9bc2a7b8c1f754）

在你了解一些RxJS背后的概念之后，下面的链接可以帮你在前进的道路上走得更远。

- “哪些静态操作符可以用来创建流？”（https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-static.md）
- “哪些实例操作符可以在流上使用？”（https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/which-instance.md）
- RxMarbles：各种流操作的交互式图解（http://staltz.com/rxmarbles）

本章由始至终都会提供RxJS的API文档链接。RxJS文档有大量很棒的示例代码，阐明了不同的流和操作符是如何工作的。



Angular必须要用RxJS吗？

不，完全不必。可观察对象只是Angular众多数据模式中的一种。想了解其他数据模式，请参见第9章。

我想给你提个醒：起初学习RxJS时会有一些烧脑。但是相信我，你终将掌握它的要领，并且这些付出都是值得的。下面是一些关于流的重要概念，会对你有所帮助。

(1) 承诺（promise）发出单个值，而流发出多个值。在应用中，流扮演着和承诺一样的角色。如果你是从回调函数转为承诺的话，会发现相对于回调函数，承诺在可读性和数据可维护性方面都有了很大的改进。同样，流也改进了承诺，可以在流上持续响应数据的变化（与此相反，承诺是一次性解决）。

(2) 命令式代码“拉取”数据，而响应式流“推送”数据。在响应式编程中，代码订阅了数据变化时接收通知，流会把数据“推送”给这些订阅者。

(3) RxJS是函数式的。如果你热衷于像`map`、`reduce`和`filter`这样的函数式操作符，那么使用RxJS时会感到很轻松；因为在某种意义上讲，数据集合和强大的函数操作符同样适用于流。

(4) 流是可组合的。可以把流想象成一个贯穿数据的操作管道。你可以订阅流中的任何部分，甚至可以组合它们来创建新的流。

10.2　聊天应用概览

在本章中，我们将使用RxJS构建聊天应用。界面截图如图10-1所示。

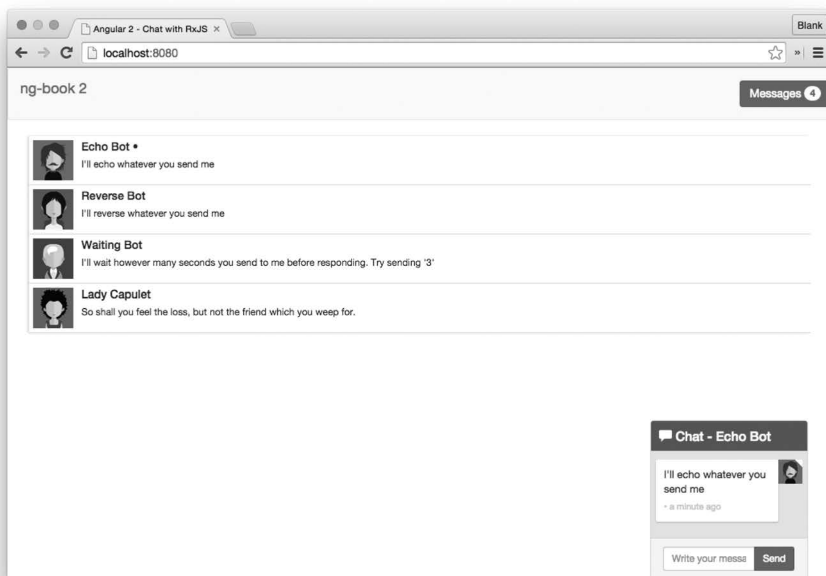


图10-1　完成后的聊天应用



我们通常会尝试在书中展现每一行代码。不过这个聊天应用有大量的活动部件，所以本章不会展现所有代码。

可以在文件夹code/rxjs/chat中找到本章的示例代码。在适当的时候，我们会告诉你在哪里可以找到你想要查看的内容。

本应用提供了几个机器人，你可以和它们聊天。先运行这些代码看看：

```
cd code/rxjs/chat
npm install
npm run go
```

然后在浏览器中打开http://localhost:8080。



如果上面的链接无法打开，请尝试这个链接：http://localhost:8080/webpack-dev-server/index.html。



一些Windows用户在这个目录下运行`npm install`时可能会遇到问题。如果遇到了，请先确保自己是在Cygwin[1]中运行这些命令行。

你在本应用中要注意以下几点：

- 你可以点击会话（thread）和一个机器人聊天；
- 机器人会根据各自的性格来回复你的消息；
- 右上角的未读消息总数会自动同步。

下面来看看本应用是如何构造的。我们有：

- 三个顶层Angular组件
- 三个数据模型
- 三个服务

让我们来逐个看看。

10.2.1 组件

将页面分解成三个顶层组件，如图10-2所示。

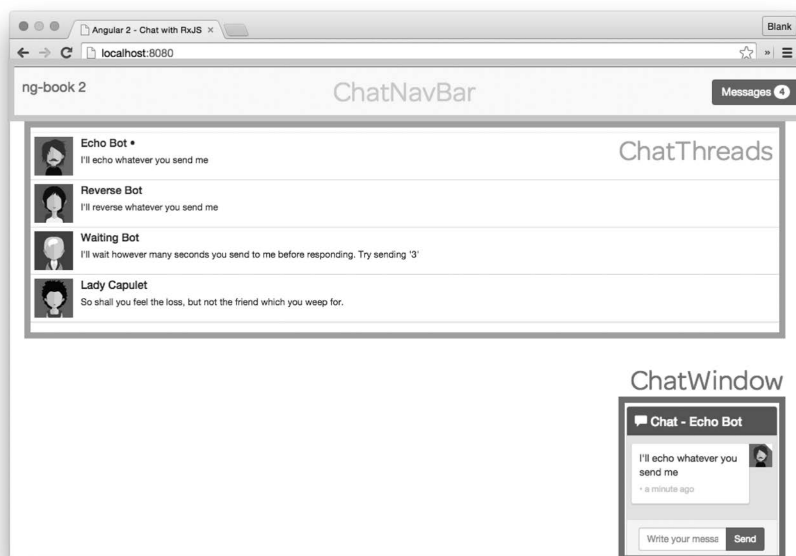


图10-2 聊天应用的顶层组件

① https://www.cygwin.com/

- ChatNavBar：包含未读消息数。
- ChatThreads：展示一个可点击的会话列表，每个会话都包含最新消息和会话头像。
- ChatWindow：展示当前会话的消息和一个用来发送新消息的输入框。

10.2.2　数据模型

本应用同样包含三个数据模型，如图10-3所示。

- User：存储聊天参与者的相关信息。
- Message：存储一条单独的信息。
- Thread：存储一组消息的集合以及一些与这次会话有关的其他数据。

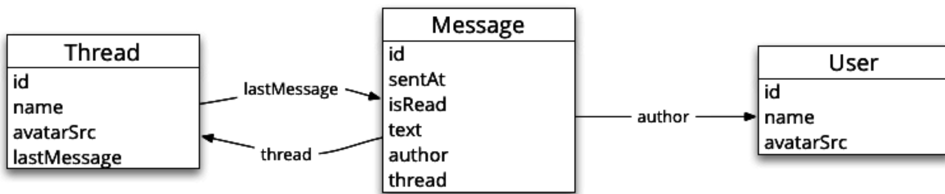


图10-3　聊天应用的数据模型

10.2.3　服务

在本应用中，每个数据模型都有其对应的服务。服务都是单例对象，有以下两个作用：

(1) 提供应用可以订阅的数据流；

(2) 提供操作符来添加或更改数据。

比如，UserService：

- 发布一个流用来通知当前用户；
- 提供一个setCurrentUser函数，用于设置当前用户（即从currentUser流发出当前用户）。

10.2.4　总结

大体上来说，本应用的数据架构很简明：

- 服务负责维护流，而流负责发出数据模型（例如Message）；
- 组件订阅这些流并按照最新的值进行渲染。

比如，ChatThreads组件订阅ThreadService中的流来获取最新的会话列表，而ChatWindow组件订阅ThreadService中的流来获取最新的消息列表。

本章其余部分将深入探讨如何使用Angular和RxJS来实现此应用。我们首先实现数据模型，然后看看如何创建服务来管理流，最后实现组件。

10.3 实现数据模型

我们先从简单的部分开始，看看数据模型。

10.3.1 `User`

`User`类很简明，有`id`、`name`和`avatarSrc`三个属性。

code/rxjs/chat/app/ts/models.ts

```
export class User {
  id: string;

  constructor(public name: string,
              public avatarSrc: string) {
    this.id = uuid();
  }
}
```



注意上面的代码，我们在构造函数中使用了TypeScript的简写方式。当指明`public name: string`时，我们是在告诉TypeScript：(1) 将`name`作为类的一个公有属性；(2) 当创建一个新的实例时，把参数的值赋给这个属性。

10.3.2 `Thread`

同样，`Thread`也是一个简单的TypeScript类。

code/rxjs/chat/app/ts/models.ts

```
export class Thread {
  id: string;
  lastMessage: Message;
  name: string;
  avatarSrc: string;

  constructor(id?: string,
              name?: string,
              avatarSrc?: string) {
    this.id = id || uuid();
    this.name = name;
    this.avatarSrc = avatarSrc;
  }
}
```

注意，我们在Thread类中保存了一个lastMessage的引用。这可以使我们在会话列表中显示最新消息。

10.3.3　Message

同样，Message也是个简单的TypeScript类，但是这里使用了一个形式略微不同的构造函数。

code/rxjs/chat/app/ts/models.ts

```
  lastMessage: Message;
```

构造函数中的这种模式允许我们使用构造函数中的关键字参数进行模拟。使用这种模式，可以使用任意的数据来创建一个新的Message，而且不用担心参数的顺序问题。比如，我们可以这样做：

```
  let msg1 = new Message();

  # or this

  let msg2 = new Message({
    text: "Hello Nate Murray!"
  })
```

看完了数据模型，我们再来看看第一个服务：UserService。

10.4　实现 UserService

UserService的意义在于提供这样一个场所：应用可以在这里了解到当前用户信息，并在当前用户发生变化时通知应用的其他部件。

我们要做的第一件事是创建一个TypeScript类，并为它加上@Injectable注解。[①]

code/rxjs/chat/app/ts/services/UserService.ts

```
export class UserService {
  // `currentUser` contains the current user
  currentUser: Subject<User> = new BehaviorSubject<User>(null);

  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
}
```

① 注意，@Injectable注解表示该类可以让Angular把其他服务注入进来，也就是说以该类作为目标。因此 在创建服务时，@Injectable注解并不是必需的，但官方的风格指南明确建议我们加上它。——译者注



我们说这个服务是可注入的，意思是可以把它注入到应用中的其他组件中。简要来说，依赖注入有两大优点：

(1) 让Angular来管理对象的生命周期；

(2) 测试组件时更容易。

我们在第8章中深入讨论了它。如果你还没有阅读第8章，现在只需要知道可以把它注入到我们的组件中就可以了，代码如下：

```
class MyComponent {
  constructor(public userService: UserService) {
    // do something with `userService` here
  }
}
```

10.4.1 currentUser 流

接下来设置一个流，用来管理当前用户。

code/rxjs/chat/app/ts/services/UserService.ts

```
  currentUser: Subject<User> = new BehaviorSubject<User>(null);
```

这里发生了很多事，我们来逐一分解：

- 定义了实例变量currentUser，它是一个Subject流；
- 更准确地说，currentUser是一个包含User的BehaviorSubject；
- 然而，这个流的初始值是null（构造函数参数）。

如果你没怎么用过RxJS的话，那么可能不知道Subject和BehaviorSubject是什么。你可以把Subject当作一个“读/写”流。



从技术上来说，Subject[1]同时了继承Observable[2]和Observer[3]。

因为消息是即时发送的，所以新的订阅者会有丢失流中最新值的风险。这是流的一个副作用，而BehaviourSubject弥补了这一点。

BehaviourSubject[4]有一个特殊的属性，用来存储最新的值。这意味着任何流的订阅者都会接收最新的值。这对于我们来说好极了，因为这意味着应用的任何部分都可以订阅UserService.currentUser流并且可以立即知道当前用户是谁。

---

① https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/subject.md

② https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md

③ https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observer.md

④ https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/subjects/behaviorsubject.md

10.4.2　设置新用户

当前用户改变时（例如登录），我们需要一个途径将新用户发布到流中。

有两种暴露API的方法可以做到这件事。

1. 直接将新用户添加到流中

更新当前用户的最直接方法就是使用UserService的实例直接发布一个新的User对象到流中，如下所示。

```
userService.subscribe((newUser) => {
  console.log('New User is: ', newUser.name);
})

// => New User is: originalUserName

let u = new User('Nate', 'anImgSrc');
userService.currentUser.next(u);

// => New User is: Nate
```



注意，这里使用了Subject的next方法来推送一个新值到流中。

这种做法的好处是可以复用流中现有的API，不需要引入任何新的代码或者API。

2. 创建setCurrentUser(newUser: User)方法

另一种更新当前用户的方法是在UserService上创建一个辅助方法，如下所示。

code/rxjs/chat/app/ts/services/UserService.ts

```
  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
```

你会注意到我们仍然在使用currentUser流的next方法。为何还要这样做呢？

这样做的价值在于，currentUser的实现与流的实现进行了解耦。通过把next方法包裹在setCurrentUser方法里，我们有一定的空间来更改UserService的实现而不至于破坏实例。

在这个例子中，我不会强烈推荐其中某一种方法，但两种方法在大型项目中的可维护性上还是有显著区别的。



第三种选项是把这些更改暴露为它们自己的流（也就是说我们把更改当前用户的这个“动作”放进流中）。我们会在下面的MessagesService中探讨这种模式。

10.4.3　UserService.ts

把所有代码整合起来，可以得到UserService的完整代码。

code/rxjs/chat/app/ts/services/UserService.ts

```
import {Injectable} from '@angular/core';
import {Subject, BehaviorSubject} from 'rxjs';
import {User} from '../models';


/**
 * UserService manages our current user
 */
@Injectable()
export class UserService {
  // `currentUser` contains the current user
  currentUser: Subject<User> = new BehaviorSubject<User>(null);

  public setCurrentUser(newUser: User): void {
    this.currentUser.next(newUser);
  }
}

export var userServiceInjectables: Array<any> = [
  UserService
];
```

10.5　MessagesService

MessagesService是这个应用的支柱。此应用中的所有消息都要流经MessagesService。

相比于UserService，MessagesService包含一些更复杂的流，它由五个流组成：三个数据管理流和两个动作流。

三个数据管理流分别是：

- newMessages，发出每条新Message并且每条只发出一次；
- messages，发出一组当前的Messages；
- updates，在messages流上执行操作。

10.5.1　newMessages 流

newMessages是一个Subject，用来发出每条新Message并且每条只发出一次。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();
```

我们还可以定义一个辅助方法来添加Message到这个流中。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
addMessage(message: Message): void {
  this.newMessages.next(message);
}
```

有这样的一个流还是很有帮助的，它可以从一个会话中获取不属于某个特殊用户的所有消息。以回声机器人（Echo Bot）为例，如图10-4所示。

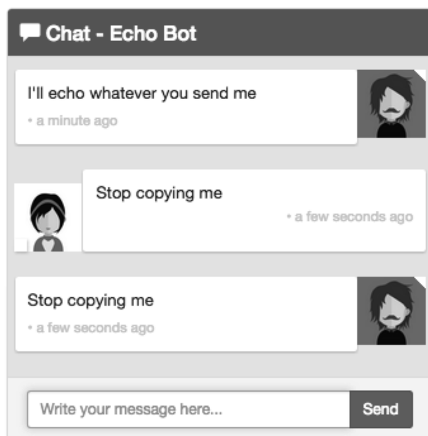


图10-4　回声机器人

当实现回声机器人时，我们不想进入一个重复机器人本身消息的死循环。

要实现这一点，我们可以订阅newMessages流并根据下面的条件过滤所有消息：

(1) 是这个会话的一部分；

(2) 不是机器人产生的。

你可以这样理解，对于一个给定的Thread，我们想要一个不包含这个User的消息流。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
  return this.newMessages
    .filter((message: Message) => {
             // belongs to this thread
      return (message.thread.id === thread.id) &&
             // and isn't authored by this user
             (message.author.id !== user.id);
    });
}
```

messagesForThreadUser接收一个Thread对象和一个User对象并返回一个经过筛选的新

Message流。筛选条件是消息属于这个Thread，而且不是由这个User写的。也就是说，这是一个在此Thread中的其他人的消息流。

10.5.2 messages 流

newMessages流发出单个的Message对象，而messages流发出一组最新的Message对象。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  messages: Observable<Message[]>;
```



类型Message[]等同于Array<Message>。另一种等价的写法是Observable<Array<Message>>。当定义messages流的类型为Observable<Message[]>时，表示这个流发出的是一个数组（Message对象的数组），而不是单个的Messages。

那么messages是如何填充的呢？为此我们需要讨论updates流和一种新的模式：操作流。

10.5.3 操作流模式

下面是操作流模式的基本理念：

- 在messages流中维护状态，它会保存一个最新的Message数组；
- 使用一个updates流，即应用于messages流的函数流。

你可以这样理解：任何updates流上的函数都会更改当前的消息列表。updates流上的函数应该接收一个Message对象列表然后返回一个Message对象列表。让我们在代码中通过创建一个接口来使这个概念形式化。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}
```

下面来定义updates流。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  updates: Subject<any> = new Subject<any>();
```

记住，updates流接收用来应用到消息列表的操作。但是如何把这些关联起来呢？实现方法如下（在MessagesService的constructor中）。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
constructor() {
  this.messages = this.updates
    // watch the updates and accumulate operations on the messages
    .scan((messages: Message[],
           operation: IMessagesOperation) => {
             return operation(messages);
           },
           initialMessages)
    // make sure we can share the most recent list of messages across anyone
```

这段代码引入了新的流函数：scan[①]。如果你熟悉函数式编程的话，scan很像reduce：它为输入流中的每个元素运行函数并累加出一个值。scan的特别之处在于，它会把每个中间过程中计算出的结果值发送出去。也就是说，它不会等到流全部完成后再发送结果值；这正是我们想要的。

当调用this.updates.scan时，我们会创建一个新的流。这个流订阅了updates流。scan内部执行的每一次，我们都会得到：

(1) 经过累加的messages流；

(2) 将要应用的新operation。

然后返回新的Message[]。

10.5.4　共享流

关于流，你需要知道的一点是它们默认是不可共享的。也就是说，如果一个订阅者从流中读取了一个值，读完后这个值就永远消失了。在这个例子中，我们想：(1) 在一些订阅者之间共享同样的流；(2) 为任何未来的订阅者重播最新的值。

要做到这点，我们使用操作符publishReplay和refCount。

- publishReplay可以让我们在多个订阅者之间共享同一个订阅，并为未来的订阅者重播n个最新的值。（参见publish[②]和replay[③]）
- refCount[④]通过对可观察对象何时发出值进行管理，使publish方法的返回值用起来更加方便。

① https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md
② https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/publish.md
③ https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/replay.md
④ https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md



等等，refCount到底是干什么的？

refCount可能有一些不太好理解，因为它涉及一个如何管理“热”的可观察对象和“冷”的可观察对象。我们不打算深入讲解它的工作原理，读者可自行阅读相关文档。

- 关于refCount的RxJS文档：https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/refcount.md
- “Rx介绍：‘热’的可观察对象和‘冷’的可观察对象”：http://www.introtorx.com/Content/ v1.0.10621.0/14_HotAndColdObservables.html#RefCount
- refCount弹珠图解：http://reactivex.io/documentation/operators/refcount.html

code/rxjs/chat/app/ts/services/MessagesService.ts

```
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .publishReplay(1)
      .refCount();
```

10.5.5 把 Message 对象添加到 messages 流中

现在我们可以把一个Message对象添加到messages流中，如下所示：

```
var myMessage = new Message(/* params here... */);

updates.next( (messages: Message[]): Message[] => {
  return messages.concat(myMessage);
})
```

我们添加了一个操作到updates流中。因为messages流订阅了updates流，所以它会应用这个操作，而操作会使用concat把我们的newMessage合并到累加的messages列表之中。



如果这里需要花费你一些时间来仔细思考，也没有关系。要是你不习惯这种编程风格的话，是会感觉有些陌生。

上面的方法有一个问题，那就是它使用起来有些繁琐。要是不用每次都写这种内部函数就好了。我们可以像下面这样做：

```
addMessage(newMessage: Message) {
  updates.next( (messages: Message[]): Message[] => {
```

```
    return messages.concat(newMessage);
  })
}

// somewhere else

var myMessage = new Message(/* params here... */);
MessagesService.addMessage(myMessage);
```

现在好一些了，但它还不是“响应式的方式”。这是因为这种创建消息的行为不能和其他流组合。（该方法也绕过了newMessage流。稍后将进行更详细的讨论。）

创建新消息的响应式做法是用一个流来接收Message对象并把它添加到消息列表中。再次声明，如果你还没有习惯这种思维方式的话，那么这对于你来说会有些陌生。下面介绍实现它的方法。

首先，我们创建一个叫作create的动作流。（动作流这个术语只是用来描述它在服务中的角色。这个流本身只是一个普通的Subject。）

code/rxjs/chat/app/ts/services/MessagesService.ts

```
// action streams
create: Subject<Message> = new Subject<Message>();
```

接下来，我们在构造函数中配置了create流。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
this.create
  .map( function(message: Message): IMessagesOperation {
    return (messages: Message[]) => {
      return messages.concat(message);
    };
  })
```

map操作符[1]和JavaScript中内置的Array.map很像，只不过它是在流上的工作。也就是说，它为流中的每一项运行函数并发出函数的返回值。

在这个例子中，我们的意思是“对于我们接收并作为输入的每个Message对象来说，都返回IMessagesOperation，它会把这个消息添加到消息列表中”。换句话说，这个流会发出一个函数，这个函数接收Message对象的列表并把这个Message对象添加到消息列表中。

现在有了create流，还有一件事要做：实际上，我们需要把create流连接到updates流。我们使用subscribe[2]来完成。

① https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/select.md
② https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/subscribe.md

code/rxjs/chat/app/ts/services/MessagesService.ts

```
    this.create
      .map( function(message: Message): IMessagesOperation {
        return (messages: Message[]) => {
          return messages.concat(message);
        };
      })
      .subscribe(this.updates);
```

我们在这里所做的就是订阅updates流来监听create流。这表示，如果create流接收了一个Message对象，那么它会发出一个IMessagesOperation；updates流会接收这个IMessages-Operation，然后把Message对象添加到messages流中。

图10-5展现了当前的情况。

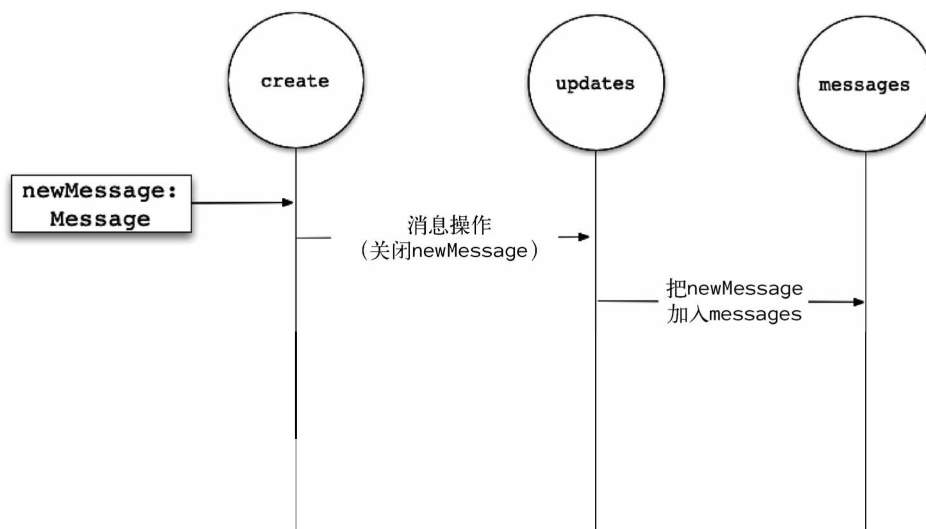


图10-5 从create流开始创建新消息

这很棒！因为它意味着我们：

(1) 从messages流中获取了当前消息列表；

(2) 获得了在当前消息列表上进行操作的一种方式（通过updates流）；

(3) 通过一个简单易用的流把创建操作放在了updates流上（通过create流）。

不论在代码的什么地方，只要想获取最新消息列表，就必须要用messages流。但是还有一个问题，我们还没有把这个流程和newMessages流关联起来。

如果有一种方式可以轻松地把这个流和任何newMessages流发出的Message关联起来，那就

太好了。事实证明这很容易。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
this.newMessages
  .subscribe(this.create);
```

现在的情况如图10-6所示。

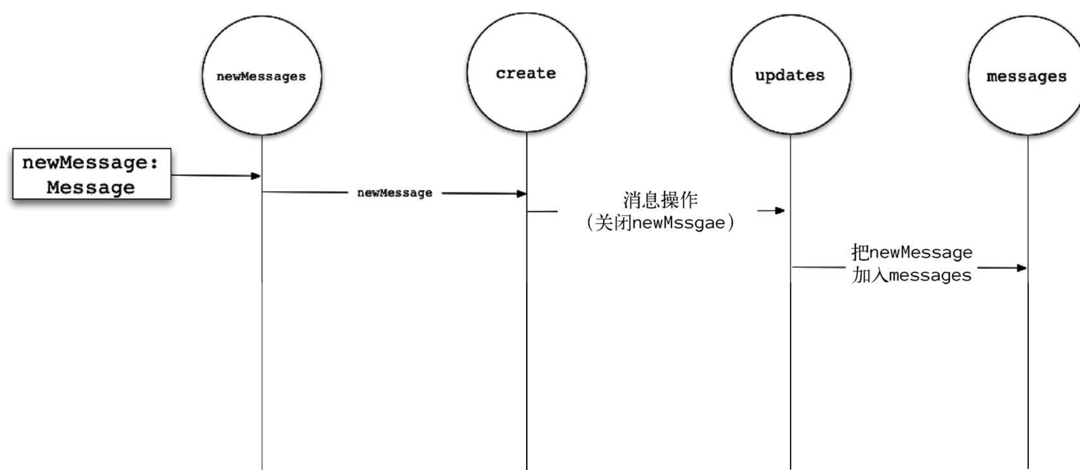


图10-6　从newMessages流开始创建新消息

现在的流程完整了！这也是两全其美的：我们能够通过订阅newMessages来获取单条消息；而如果只想要最新的消息列表，我们可以订阅messages流。



这里需要指出这个设计的一些影响：如果你直接订阅了newMessages流，必须要注意变化可能发生在下游。这里有三点需要考虑。
第一，显然不会有任何下游的更新应用于Message。
第二，在这个案例中，我们的Message对象是**可变的**。如果你订阅newMessages流并保存了Message的引用，那么这个Message的属性可能会产生变化。
第三，如果想利用Message的可变性，你可能无法做到。考虑这种情况：我们可以在updates流队列上增加一个操作，此操作复制每个Message然后改变这个副本。（与我们现在的做法相比，这应该是更好的设计。）在这个例子中，你不能依赖任何从newMessages流直接发出的Message，因为它们是可以改变的。
尽管如此，只要你记住这些注意事项，就应该不会有太大麻烦。

10.5.6　完整的 MessagesService

完整的MessagesService代码如下。

code/rxjs/chat/app/ts/services/MessagesService.ts

```
import {Injectable} from '@angular/core';
import {Subject, Observable} from 'rxjs';
import {User, Thread, Message} from '../models';

let initialMessages: Message[] = [];

interface IMessagesOperation extends Function {
  (messages: Message[]): Message[];
}

@Injectable()
export class MessagesService {
  // a stream that publishes new messages only once
  newMessages: Subject<Message> = new Subject<Message>();

  // `messages` is a stream that emits an array of the most up to date messages
  messages: Observable<Message[]>;

  // `updates` receives _operations_ to be applied to our `messages`
  // it's a way we can perform changes on *all* messages (that are currently
  // stored in `messages`)
  updates: Subject<any> = new Subject<any>();

  // action streams
  create: Subject<Message> = new Subject<Message>();
  markThreadAsRead: Subject<any> = new Subject<any>();

  constructor() {
    this.messages = this.updates
      // watch the updates and accumulate operations on the messages
      .scan((messages: Message[],
             operation: IMessagesOperation) => {
               return operation(messages);
             },
            initialMessages)
      // make sure we can share the most recent list of messages across anyone
      // who's interested in subscribing and cache the last known list of
      // messages
      .publishReplay(1)
      .refCount();

      // `create` takes a Message and then puts an operation (the inner function)
      // on the `updates` stream to add the Message to the list of messages.
      //
      // That is, for each item that gets added to `create` (by using `next`)
      // this stream emits a concat operation function.
      //
      // Next we subscribe `this.updates` to listen to this stream, which means
      // that it will receive each operation that is created
      //
      // Note that it would be perfectly acceptable to simply modify the
      // "addMessage" function below to simply add the inner operation function to
```

```
      // the update stream directly and get rid of this extra action stream
      // entirely. The pros are that it is potentially clearer. The cons are that
      // the stream is no longer composable.
      this.create
        .map( function(message: Message): IMessagesOperation {
          return (messages: Message[]) => {
            return messages.concat(message);
          };
        })
        .subscribe(this.updates);

      this.newMessages
        .subscribe(this.create);

      // similarly, `markThreadAsRead` takes a Thread and then puts an operation
      // on the `updates` stream to mark the Messages as read
      this.markThreadAsRead
        .map( (thread: Thread) => {
          return (messages: Message[]) => {
            return messages.map( (message: Message) => {
              // note that we're manipulating `message` directly here. Mutability
              // can be confusing and there are lots of reasons why you might want
              // to, say, copy the Message object or some other 'immutable' here
              if (message.thread.id === thread.id) {
                message.isRead = true;
              }
              return message;
            });
          };
        })
        .subscribe(this.updates);

    }

    // an imperative function call to this action stream
    addMessage(message: Message): void {
      this.newMessages.next(message);
    }

    messagesForThreadUser(thread: Thread, user: User): Observable<Message> {
      return this.newMessages
        .filter((message: Message) => {
                 // belongs to this thread
          return (message.thread.id === thread.id) &&
                 // and isn't authored by this user
                 (message.author.id !== user.id);
        });
    }
  }

  export var messagesServiceInjectables: Array<any> = [
    MessagesService
  ];
```

10.5.7 试用 MessagesService

如果你还没有完全理解，那么现在是个打开代码并随意尝试MessagesService的好时机，来感受一下它是如何运作的。在test/services/MessagesService.spec.ts中有一个示例，可以直接拿来使用。



要运行这个项目的测试，可以打开终端，然后输入以下代码：

```
cd /path/to/code/rxjs/chat // <-- your path will vary
npm install
karma start
```

首先创建一些数据模型的实例。

code/rxjs/chat/test/services/MessagesService.spec.ts

```
import {MessagesService} from '../../app/ts/services/services';
import {Message, User, Thread} from '../../app/ts/models';

describe('MessagesService', () => {
  it('should test', () => {

    let user: User = new User('Nate', '');
    let thread: Thread = new Thread('t1', 'Nate', '');
    let m1: Message = new Message({
      author: user,
      text: 'Hi!',
      thread: thread
    });

    let m2: Message = new Message({
      author: user,
      text: 'Bye!',
      thread: thread
    });
```

接下来，订阅几个流。

code/rxjs/chat/test/services/MessagesService.spec.ts

```
    let messagesService: MessagesService = new MessagesService();

    // listen to each message indivdually as it comes in
    messagesService.newMessages
      .subscribe( (message: Message) => {
        console.log('=> newMessages: ' + message.text);
      });

    // listen to the stream of most current messages
    messagesService.messages
      .subscribe( (messages: Message[]) => {
        console.log('=> messages: ' + messages.length);
      });
```

```
    messagesService.addMessage(m1);
    messagesService.addMessage(m2);

    // => messages: 1
    // => newMessages: Hi!
    // => messages: 2
    // => newMessages: Bye!
  });


});
```

注意，尽管我们先订阅了newMessages并且newMessages是通过addMessage方法直接调用的，但是messages流的订阅先输出了日志。原因就是messages流订阅newMessages流早于测试代码中的订阅（当MessagesService实例化时）。（你不应该依赖于代码中单独的流的顺序，但是它为什么以这种方式运行是值得思考的。）

尝试使用MessagesService并感受一下这些流是如何工作的。我们将在下节中使用它们来构建ThreadsService。

10.6　ThreadsService

在ThreadsService中将定义四个流，它们分别发出：

(1) 当前一组Thread的映射（threads流）；

(2) 按时间逆序排列的Thread列表（orderedthreads流）；

(3) 当前已选的Thread（currentThread流）；

(4) 当前已选Thread的Message列表（currentThreadMessages流）。

下面来讨论如何构建这里的每一个流。在这个过程中，我们还将学习更多关于RxJS的知识。

10.6.1　当前一组 Thread 的映射（threads 流）

我们先来定义ThreadsService类和用来发出Thread的实例变量。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
import {Injectable} from '@angular/core';
import {Subject, BehaviorSubject, Observable} from 'rxjs';
import {Thread, Message} from '../models';
import {MessagesService} from './MessagesService';
import * as _ from 'underscore';

@Injectable()
export class ThreadsService {
```

```
// `threads` is a observable that contains the most up to date list of threads
threads: Observable<{ [key: string]: Thread }>;
```

注意，这个流会发出一个映射（即一个对象），将Thread的id作为string键，Thread本身作为值。

要创建一个用来维护当前会话列表的流，我们先附加到messagesService.messages流。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
threads: Observable<{ [key: string]: Thread }>;
```

回忆一下，每次把一个新的Message对象添加到流时，messages流都会发出一个当前Message对象的数组。我们要查看每个Message对象并返回唯一的Threads列表。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;
```

注意，每次都会创建一个新的threads列表。这样做的原因是，我们可能会彻底删除一些消息（例如离开对话）。因为每次我们都重新计算会话列表，所以自然而然地“删除”了没有消息的会话。

在会话列表中，我们想通过使用Thread中的最新Message来显示聊天预览。



图10-7　带有聊天预览功能的会话列表

要做到这一点，我们在每个Thread中都保存了最新的Message。通过比较sentAt时间就可以知道哪个Message是最新的。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
        // Cache the most recent message for each thread
        let messagesThread: Thread = threads[message.thread.id];
        if (!messagesThread.lastMessage ||
            messagesThread.lastMessage.sentAt < message.sentAt) {
          messagesThread.lastMessage = message;
        }
      });
      return threads;
    });
```

把所有代码整合起来，threads流看起来如下所示。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
            message.thread;

        // Cache the most recent message for each thread
        let messagesThread: Thread = threads[message.thread.id];
        if (!messagesThread.lastMessage ||
            messagesThread.lastMessage.sentAt < message.sentAt) {
          messagesThread.lastMessage = message;
        }
      });
      return threads;
    });
```

试用ThreadsService

我们来试试ThreadsService。首先创建一些要用的数据模型。

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
import {MessagesService, ThreadsService} from '../../app/ts/services/services';
import {Message, User, Thread} from '../../app/ts/models';
import * as _ from 'underscore';

describe('ThreadsService', () => {
  it('should collect the Threads from Messages', () => {

    let nate: User = new User('Nate Murray', '');
    let felipe: User = new User('Felipe Coury', '');

    let t1: Thread = new Thread('t1', 'Thread 1', '');
    let t2: Thread = new Thread('t2', 'Thread 2', '');

    let m1: Message = new Message({
      author: nate,
```

```
      text: 'Hi!',
      thread: t1
    });

    let m2: Message = new Message({
      author: felipe,
      text: 'Where did you get that hat?',
      thread: t1
    });

    let m3: Message = new Message({
      author: nate,
      text: 'Did you bring the briefcase?',
      thread: t2
    });
```

创建服务的一个实例。

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
    let messagesService: MessagesService = new MessagesService();
    let threadsService: ThreadsService = new ThreadsService(messagesService);
```



注意，这里把messagesService作为参数传给了ThreadsService的构造函数。我们通常让依赖注入系统来处理这些，但在测试中可以自己提供依赖关系。

我们订阅threads流并把通过流的内容打印出来。

code/rxjs/chat/test/services/ThreadsService.spec.ts

```
    let threadsService: ThreadsService = new ThreadsService(messagesService);

    threadsService.threads
      .subscribe( (threadIdx: { [key: string]: Thread }) => {
        let threads: Thread[] = _.values(threadIdx);
        let threadNames: string = _.map(threads, (t: Thread) => t.name)
                                 .join(', ');
        console.log(`=> threads (${threads.length}): ${threadNames} `);
      });

    messagesService.addMessage(m1);
    messagesService.addMessage(m2);
    messagesService.addMessage(m3);

    // => threads (1): Thread 1
    // => threads (1): Thread 1
    // => threads (2): Thread 1, Thread 2

  });
});
```

10.6.2　按时间逆序排列的 Thread 列表（orderedthreads 流）

threads流给了我们一个映射，作为会话列表的一个“索引”。但是我们想让会话视图根据最新消息的时间来排序，如图10-8所示。



图10-8　按时间逆序排列的会话

创建一个新的流，它返回一个按最新Message时间排序的Thread数组。

我们首先定义orderedThreads并把它作为一个实例属性。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
// `orderedThreads` contains a newest-first chronological list of threads
orderedThreads: Observable<Thread[]>;
```

接下来，在constructor中通过订阅threads流并按最新消息时间排序定义orderedThreads。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
    this.orderedThreads = this.threads
      .map((threadGroups: { [key: string]: Thread }) => {
        let threads: Thread[] = _.values(threadGroups);
        return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
      });
```

10.6.3　当前已选的 Thread（currentThread 流）

我们的应用需要知道当前已选的Thread是哪个。这让我们知道：

(1) 哪个会话应该在消息窗口显示；

(2) 会话列表中的哪个会话应该被标记为当前会话（如图10-9所示）。

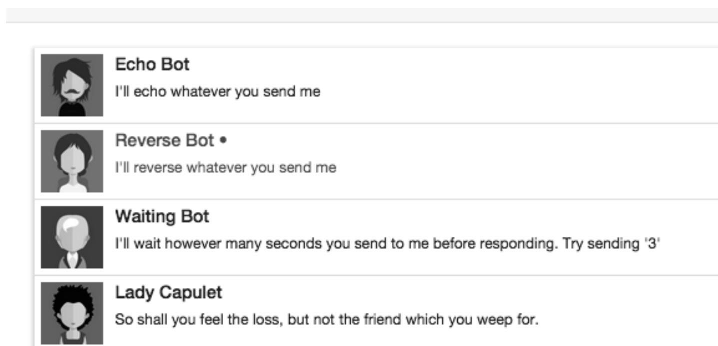


图10-9 使用·符号表示当前会话

创建一个BehaviorSubject并把它保存为currentThread流。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
// `currentThread` contains the currently selected thread
currentThread: Subject<Thread> =
  new BehaviorSubject<Thread>(new Thread());
```

注意，这里分配了一个空的Thread作为默认值。我们不再需要对currentThread进行更多配置了。

1. 设置当前会话

要设置当前会话，currentThread流可以选择下面的其中一个方法：

(1) 直接通过next方法提交新会话；

(2) 添加一个辅助函数提交新会话。

我们定义一个辅助函数setCurrentThread，可以使用它来设置下一个会话。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
setCurrentThread(newThread: Thread): void {
  this.currentThread.next(newThread);
}
```

2. 标记当前会话为已读

我们想要记录未读消息数量。如果切换到一个新Thread，要把那个Thread中的所有Message都标记为已读。我们拥有做到这些所需的工具：

(1) messagesService.makeThreadAsRead接收一个Thread，然后把这个Thread中的所有Message都标记为已读；

(2) currentThread流发出单个的Thread，它代表当前Thread。

要做的就是把它们关联起来。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
this.currentThread.subscribe(this.messagesService.markThreadAsRead);
```

10.6.4　当前已选 Thread 的 Message 列表（currentThreadMessages 流）

现在有了当前已选会话，需要确保显示这个Thread的Message列表（如图10-10所示）。

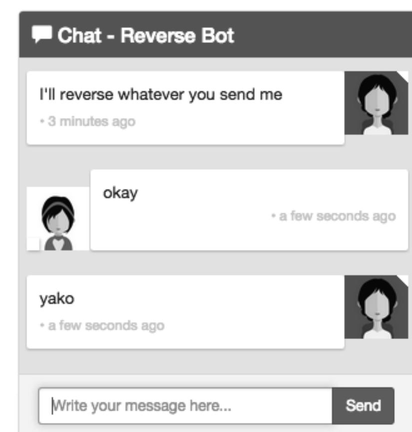


图10-10　当前消息列表来自反转机器人（Reverse Bot）

它的实现比表面上看起来要复杂一些。我们这样来实现它：

```
var theCurrentThread: Thread;

this.currentThread.subscribe((thread: Thread) => {
  theCurrentThread = thread;
})

this.currentThreadMessages.map(
  (mesages: Message[]) => {
    return _.filter(messages,
      (message: Message) => {
        return message.thread.id == theCurrentThread.id;
      })
  })
```

这种方法有什么问题？如果currentThread改变了，而currentThreadMessages完全不知道，那么currentThreadMessages就是一个过时了的消息列表！

如果颠倒一下呢？在一个变量中保存当前消息列表，然后订阅currentThread流的变化，会发生什么呢？还是会有同样的问题，只是这次我们知道会话变化，但是不知道有新消息进来。

如何解决这个问题呢？

原来，RxJS有一组操作符用来合并多个流。在这个例子中，我们想说的是“如果currentThread和messagesService.messages中的任何一个改变了，那么就要发出一些东西”。为此，我们使用combineLatest操作符[①]。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
this.currentThreadMessages = this.currentThread
  .combineLatest(messagesService.messages,
                (currentThread: Thread, messages: Message[]) => {
```

当合并两个流时，会有一个先到达，不能保证在两个流上都有值，所以需要检查以确保有我们所需要的；否则就会返回一个空列表。

现在有了当前会话和消息列表，就可以过滤出我们想要的消息了。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
this.currentThreadMessages = this.currentThread
  .combineLatest(messagesService.messages,
                (currentThread: Thread, messages: Message[]) => {
    if (currentThread && messages.length > 0) {
      return _.chain(messages)
        .filter((message: Message) =>
                (message.thread.id === currentThread.id))
```

还有一个细节：既然我们已经找到了当前会话的消息，把这些消息标记为已读就是很方便的。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
      return _.chain(messages)
        .filter((message: Message) =>
                (message.thread.id === currentThread.id))
        .map((message: Message) => {
          message.isRead = true;
          return message; })
        .value();
```



关于是否应该在这里把消息标记为已读是有争议的。标记为已读的最大缺点就是我们更改了对象本身，而本质上这是一个“只读”会话。也就是说，这是一个有副作用的读操作，一般不应该使用。尽管如此，本应用中的currentThreadMessages流只作用于currentThread流，而currentThread流应始终把它的消息标记为已读。不过，我通常不推荐“有副作用的读操作”模式。

把所有代码整合起来，currentThreadMessages看起来是这样的。

① https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/combinelatestproto.md

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
    this.currentThreadMessages = this.currentThread
      .combineLatest(messagesService.messages,
                     (currentThread: Thread, messages: Message[]) => {
        if (currentThread && messages.length > 0) {
          return _.chain(messages)
            .filter((message: Message) =>
                    (message.thread.id === currentThread.id))
            .map((message: Message) => {
              message.isRead = true;
              return message; })
            .value();
        } else {
          return [];
        }
      });
```

10.6.5　完整的 ThreadsService

ThreadService完整代码如下所示。

code/rxjs/chat/app/ts/services/ThreadsService.ts

```
import {Injectable} from '@angular/core';
import {Subject, BehaviorSubject, Observable} from 'rxjs';
import {Thread, Message} from '../models';
import {MessagesService} from './MessagesService';
import * as _ from 'underscore';

@Injectable()
export class ThreadsService {

  // `threads` is a observable that contains the most up to date list of threads
  threads: Observable<{ [key: string]: Thread }>;

  // `orderedThreads` contains a newest-first chronological list of threads
  orderedThreads: Observable<Thread[]>;

  // `currentThread` contains the currently selected thread
  currentThread: Subject<Thread> =
    new BehaviorSubject<Thread>(new Thread());

  // `currentThreadMessages` contains the set of messages for the currently
  // selected thread
  currentThreadMessages: Observable<Message[]>;

  constructor(private messagesService: MessagesService) {

    this.threads = messagesService.messages
      .map( (messages: Message[]) => {
        let threads: {[key: string]: Thread} = {};
        // Store the message's thread in our accumulator `threads`
        messages.map((message: Message) => {
          threads[message.thread.id] = threads[message.thread.id] ||
```

```
          message.thread;

        // Cache the most recent message for each thread
        let messagesThread: Thread = threads[message.thread.id];
        if (!messagesThread.lastMessage ||
            messagesThread.lastMessage.sentAt < message.sentAt) {
          messagesThread.lastMessage = message;
        }
      });
      return threads;
    });

  this.orderedThreads = this.threads
    .map((threadGroups: { [key: string]: Thread }) => {
      let threads: Thread[] = _.values(threadGroups);
      return _.sortBy(threads, (t: Thread) => t.lastMessage.sentAt).reverse();
    });

  this.currentThreadMessages = this.currentThread
    .combineLatest(messagesService.messages,
                   (currentThread: Thread, messages: Message[]) => {
      if (currentThread && messages.length > 0) {
        return _.chain(messages)
          .filter((message: Message) =>
                  (message.thread.id === currentThread.id))
          .map((message: Message) => {
            message.isRead = true;
            return message; })
          .value();
      } else {
        return [];
      }
    });

  this.currentThread.subscribe(this.messagesService.markThreadAsRead);
 }

 setCurrentThread(newThread: Thread): void {
   this.currentThread.next(newThread);
 }

}

export var threadsServiceInjectables: Array<any> = [
  ThreadsService
];
```

10.7 总结

数据模型和服务已经完成！现在，我们拥有了连接到视图组件所需要的一切！在下章中，我们将构建三个重要的组件，用来渲染页面并和本章所创建的流进行交互。

第11章 使用可观察对象的数据架构，第2部分：视图组件

11.1 构建视图：顶层组件ChatApp

现在把注意力转向应用并来完成视图组件。



为了简洁以及节省空间起见，本章会省去一些import声明、CSS和一些其他类似的代码行。如果你对这些细节的每一行代码都感兴趣的话，可以打开示例代码，那里囊括了运行程序所需要的一切。

首先要做的就是创建顶层组件chat-app。

正如之前讨论过的，页面会被分解成三个顶层组件（如图11-1所示）。

- ChatNavBar：包含未读消息数。
- ChatThreads：展示一个可点击的会话列表，每个会话都包含最新消息和会话头像。
- ChatWindow：展示当前会话的消息和一个用来发送新消息的输入框。

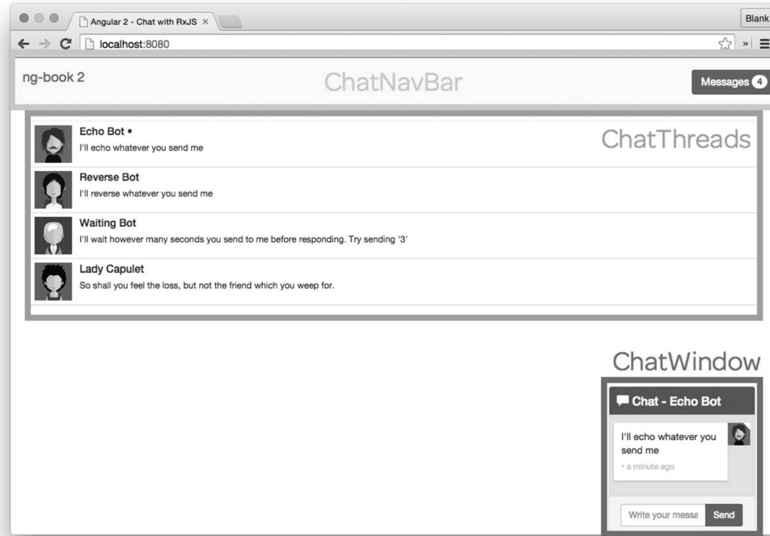


图11-1　聊天应用的顶层组件

下面是组件的代码。

code/rxjs/chat/app/ts/app.ts

```
@Component({
  selector: 'chat-app',
  template: `
  <div>
    <nav-bar></nav-bar>
    <div class="container">
      <chat-threads></chat-threads>
      <chat-window></chat-window>
    </div>
  </div>
  `
})
class ChatApp {
  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService,
              private userService: UserService) {
    ChatExampleData.init(messagesService, threadsService, userService);
  }
}

@NgModule({
  declarations: [
    ChatApp,
    ChatNavBar,
    ChatThreads,
```

11

```
    ChatThread,
    ChatWindow,
    ChatMessage,
    utilInjectables
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  bootstrap: [ ChatApp ],
  providers: [ servicesInjectables ]
})
export class ChatAppModule {}

platformBrowserDynamic().bootstrapModule(ChatAppModule);
```

注意constructor，在这个构造函数中我们要注入三个服务：MessagesService、ThreadsService和UserService。我们使用这些服务来初始化示例数据。



如果你对示例数据感兴趣的话，可以在code/rxjs/chat/app/ts/ChatExampleData.ts中找到它。

11.2　ChatThreads 组件

接下来，我们在ChatThreads组件中构建会话列表。



图11-2　按时间排序的会话列表

selector非常直观，我们要匹配chat-threads元素。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  selector: 'chat-threads',
```

11.2.1 ChatThreads 控制器

下面看看组件的控制器ChatThreads类。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
export class ChatThreads {
  threads: Observable<any>;

  constructor(private threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

我们在这里注入了ThreadsService，然后保存了orderedThreads的引用。

11.2.2 ChatThreads 的 template

最后，我们来看一下template及其配置。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  selector: 'chat-threads',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <!-- conversations -->
    <div class="row">
      <div class="conversation-wrap">

        <chat-thread
             *ngFor="let thread of threads | async"
             [thread]="thread">
        </chat-thread>

      </div>
    </div>
  `
```

这里需要注意的是，使用async管道的ngFor指令、ChangeDetectionStrategy和ChatThread组件。

ChatThread组件（在标记中匹配chat-thread）将展现聊天会话的视图。我们稍后就会来定义它。

ngFor遍历threads属性并把值通过输入属性[thread]传给ChatThread组件。但你可能注意到*ngFor中出现了新东西：async管道。

async是通过AsyncPipe实现的，它可以让我们在视图中使用RxJS的Observable。async的强大之处在于可以让我们像使用同步集合一样来使用异步可观察对象。这个特性极其方便并且非常棒。

在这个组件中，我们指定了一个特定的changeDetection。Angular提供一个灵活高效的变更探测系统。它的好处之一就是如果一个组件拥有不变的或者可观察的绑定，那么我们可以向变更探测系统发送提示，让应用高效地运行。

在这个例子中，Angular不再观察Thread数组的变化；取而代之的是订阅可观察对象threads的变化，并且在一个新的事件发出后触发更新。

下面是完整的ChatThreads组件。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  selector: 'chat-threads',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <!-- conversations -->
    <div class="row">
      <div class="conversation-wrap">

        <chat-thread
             *ngFor="let thread of threads | async"
             [thread]="thread">
        </chat-thread>

      </div>
    </div>
  `
})
export class ChatThreads {
  threads: Observable<any>;

  constructor(private threadsService: ThreadsService) {
    this.threads = threadsService.orderedThreads;
  }
}
```

11.3　单个 ChatThread 组件

下面来看一下ChatThread组件，它用来展示单个会话。我们先从@Component开始。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  inputs: ['thread'],
  selector: 'chat-thread',
  template: `
  <div class="media conversation">
    <div class="pull-left">
      <img class="media-object avatar"
           src="{{thread.avatarSrc}}">
    </div>
    <div class="media-body">
```

```
      <h5 class="media-heading contact-name">{{thread.name}}
        <span *ngIf="selected">&bull;</span>
      </h5>
      <small class="message-preview">{{thread.lastMessage.text}}</small>
    </div>
    <a (click)="clicked($event)" class="div-link">Select</a>
  </div>
  `
})
```

稍后再回来看template，我们先来看看组件定义的控制器。

11.3.1 ChatThread 控制器和 ngOnInit

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
export class ChatThread implements OnInit {
  thread: Thread;
  selected: boolean = false;

  constructor(private threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```


注意这里实现了一个新的接口：OnInit。Angular组件可以声明它们监听了某些生命周期事件。第14章会进一步讨论生命周期事件。

在这个例子中，因为我们已经声明实现了OnInit，所以当组件第一次检查变化后就会调用组件中的ngOnInit方法。

使用ngOnInit的一个关键原因在于输入属性thread在constructor中是获取不到的。

在上面可以看到，我们在ngOnInit中订阅了threadsService.currentThread。如果currentThread匹配组件中的thread属性，那么就把selected属性设置为true。（如果不匹配，就把selected属性设置为false。）

我们还设置了一个事件处理器clicked，用来处理选择当前会话的事件。在template中（参

见11.3.2节），我们会把会话视图上的点击和clicked()绑定。如果触发了clicked()，就告诉threadsService要把组件的Thread设置成当前会话设置。

11.3.2　ChatThread 的 template

下面是template的代码。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
template: `
<div class="media conversation">
  <div class="pull-left">
    <img class="media-object avatar"
         src="{{thread.avatarSrc}}">
  </div>
  <div class="media-body">
    <h5 class="media-heading contact-name">{{thread.name}}
      <span *ngIf="selected">&bull;</span>
    </h5>
    <small class="message-preview">{{thread.lastMessage.text}}</small>
  </div>
  <a (click)="clicked($event)" class="div-link">Select</a>
</div>
`
```

注意这里有一些简单的绑定，如{{thread.avatarSrc}}、{{thread.name}}和{{thread.lastMessage.text}}。

我们还用*ngIf来显示符号•，只有已选择的会话才会显示。

最后绑定了(click)事件来调用clicked()处理器。注意，调用clicked时传入了参数$event，这是一个用来描述事件的特殊变量，由Angular提供。我们在clicked处理器中通过调用方法event.preventDefault();使用了$event变量。这可以确保我们不会跳转至其他页面。

11.3.3　ChatThread 的完整代码

下面是完整的ChatThread组件。

code/rxjs/chat/app/ts/components/ChatThreads.ts

```
@Component({
  inputs: ['thread'],
  selector: 'chat-thread',
  template: `
  <div class="media conversation">
    <div class="pull-left">
      <img class="media-object avatar"
           src="{{thread.avatarSrc}}">
    </div>
```

```
      <div class="media-body">
        <h5 class="media-heading contact-name">{{thread.name}}
          <span *ngIf="selected">&bull;</span>
        </h5>
        <small class="message-preview">{{thread.lastMessage.text}}</small>
      </div>
      <a (click)="clicked($event)" class="div-link">Select</a>
    </div>
    `
})
export class ChatThread implements OnInit {
  thread: Thread;
  selected: boolean = false;

  constructor(private threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.threadsService.currentThread
      .subscribe( (currentThread: Thread) => {
        this.selected = currentThread &&
          this.thread &&
          (currentThread.id === this.thread.id);
      });
  }

  clicked(event: any): void {
    this.threadsService.setCurrentThread(this.thread);
    event.preventDefault();
  }
}
```

11.4　ChatWindow 组件

ChatWindow是此应用中最复杂的组件（如图11-3所示）。我们一步一步来完成它。

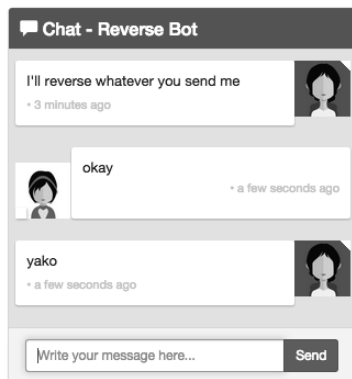


图11-3　聊天窗口

首先从定义@Component开始。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  selector: 'chat-window',
  changeDetection: ChangeDetectionStrategy.OnPush,
```

11.4.1　ChatWindow 组件类属性

ChatWindow类有四个属性。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
export class ChatWindow implements OnInit {
  messages: Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;
```

图11-4表明了每一个属性在何处使用。

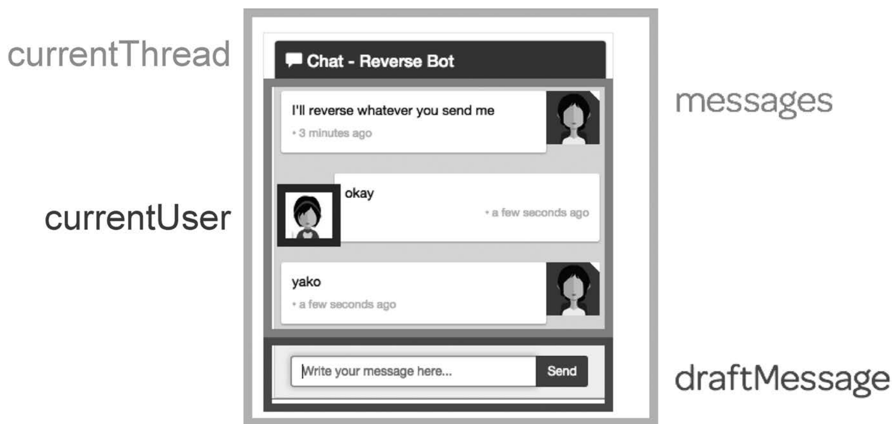


图11-4　聊天窗口的属性

我们会在constructor中注入四样东西。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService,
              private userService: UserService,
              private el: ElementRef) {
  }
```

前面的三个都是我们创建的服务。最后的el是一个ElementRef对象，可以获取当前的宿主DOM元素。当创建和接收新消息时，我们会使用它把聊天窗口滚动到底部。



请记住：通过在构造函数中使用public messagesService: MessagesService，我们在注入MessagesService的同时创建了一个实例变量，这个变量可以在类中通过this.messagesService来使用。

11.4.2 ChatWindow 的 ngOnInit

我们会把这个组件的初始化放在ngOnInit中。在这里主要要做的是，对于可以改变组件属性的可观察对象创建订阅。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
ngOnInit(): void {
  this.messages = this.threadsService.currentThreadMessages;

  this.draftMessage = new Message();
```

首先，我们会把currentThreadMessages保存到messages属性中。接下来，创建一个空的Message实例作为draftMessage属性的默认值。

当发送一条新消息的时候，需要确保这个Message保存了一份将要发送的Thread的引用。因为这个要发送的会话会成为当前会话，所以我们保存了当前已选会话的引用。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
this.threadsService.currentThread.subscribe(
  (thread: Thread) => {
    this.currentThread = thread;
  });
```

我们还希望新消息是由当前用户发送的，所以对currentUser做了同样的事。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
this.userService.currentUser
  .subscribe(
    (user: User) => {
      this.currentUser = user;
    });
```

11.4.3 ChatWindow 的 sendMessage

既然讨论到这了，那就来实现sendMessage方法，它可以发送一条新消息。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
sendMessage(): void {
  let m: Message = this.draftMessage;
  m.author = this.currentUser;
  m.thread = this.currentThread;
```

```
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
  }
```

sendMessage函数先获取draftMessage并用组件属性设置了author和thread属性。每条已发送的信息其实都已经被读过了（因为是我们写的），所以将其标记为已读。

注意，我们没有更新draftMessage的文本。这是因为很快就会将draftMessage的文本值绑定到视图中。

当draftMessage属性更新后，我们将它发送给messagesService，然后创建一个新的Message对象并赋值给this.draftMessage。这样做是为了确保不会改变已发送出去的消息。

11.4.4　ChatWindow 的 onEnter

在视图中，我们希望在下面两种场景发送消息：

(1) 用户点击Send按钮；

(2) 用户敲击回车键。

我们定义一个函数来处理这两种事件。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }
```

11.4.5　ChatWindow 的 scrollToBottom

当发送或者收到一条新消息时，我们想滚动到聊天窗口底部。为此要设置宿主元素的scrollTop属性。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
  scrollToBottom(): void {
    let scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }
```

现在有了滚动到底部的函数，还需要确保在恰当的时间调用它。回到ngOnInit方法中，订阅currentThreadMessages的消息集合并在得到一条新消息的时候滚动到底部。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
    this.messages
```

```
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
  }
```



为什么要使用setTimeout？

如果我们得到新消息时立即调用scrollToBottom，那么滚动到底部的动作就是在新消息渲染完成之前执行的。使用setTimeout可以告诉JavaScript我们要在当前执行队列完成后再运行这个函数。该函数会在组件渲染完成**之后**执行，这正是我们想要的效果。

11.4.6 ChatWindow 的 template

template的开头部分看起来应该很眼熟，我们定义了一些标记和面板标题。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  selector: 'chat-window',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>
```

接下来显示消息列表。这里使用带async管道的ngFor指令来遍历消息列表。我们很快就会讲解单个的chat-message组件。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
              <div class="panel-body msg-container-base">
                <chat-message
                     *ngFor="let message of messages | async"
                     [message]="message">
```

11

```
            </chat-message>
          </div>
```

最后是消息输入框和各个结束标签。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
        <div class="panel-footer">
          <div class="input-group">
            <input type="text"
                   class="chat-input"
                   placeholder="Write your message here..."
                   (keydown.enter)="onEnter($event)"
                   [(ngModel)]="draftMessage.text" />
            <span class="input-group-btn">
              <button class="btn-chat"
                (click)="onEnter($event)"
                >Send</button>
            </span>
          </div>
        </div>

      </div>
    </div>
  </div>
</div>
`
```

消息输入框是视图中最有意思的部分，我们来看看其中两个有趣的属性：(keydown.enter)和[(ngModel)]。

11.4.7　处理键盘动作

Angular提供了一种简明的方式来处理键盘动作：在元素上绑定事件。在这个例子中，我们绑定了keydown.enter。这表示如果用户按下回车键，就会调用表达式里的函数onEnter($event)。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
            <input type="text"
                   class="chat-input"
                   placeholder="Write your message here..."
                   (keydown.enter)="onEnter($event)"
                   [(ngModel)]="draftMessage.text" />
```

11.4.8　使用 ngModel

如前所述，Angular并没有把双向绑定作为一般模式。然而，组件和组件对应视图之间的双向绑定是非常有用的。只要把双向绑定的副作用限制在组件之中，那么保持一个组件属性和视图中同步还是非常方便的。

在这个例子中，我们在输入框的值和draftMessage.text之间建立了一个双向绑定。如果在输入框中输入文字，draftMessage.text就会自动设置为输入的文字。同样，如果在代码中更新draftMessage.text，那么视图中输入框的值也会随之改变。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
<input type="text"
       class="chat-input"
       placeholder="Write your message here..."
       (keydown.enter)="onEnter($event)"
       [(ngModel)]="draftMessage.text" />
```

11.4.9 点击 Send 按钮

在Send按钮上将(click)属性绑定到组件中的onEnter函数。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
<span class="input-group-btn">
  <button class="btn-chat"
    (click)="onEnter($event)"
    >Send</button>
</span>
```

11.4.10 完整的 ChatWindow 组件

下面是ChatWindow组件的完整代码清单。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  selector: 'chat-window',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>
```

```
          <div class="panel-body msg-container-base">
            <chat-message
                 *ngFor="let message of messages | async"
                 [message]="message">
            </chat-message>
          </div>

          <div class="panel-footer">
            <div class="input-group">
              <input type="text"
                     class="chat-input"
                     placeholder="Write your message here..."
                     (keydown.enter)="onEnter($event)"
                     [(ngModel)]="draftMessage.text" />
              <span class="input-group-btn">
                <button class="btn-chat"
                  (click)="onEnter($event)"
                  >Send</button>
              </span>
            </div>
          </div>

        </div>
      </div>
    </div>
  </div>
  `
})
export class ChatWindow implements OnInit {
  messages: Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;

  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService,
              private userService: UserService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    this.draftMessage = new Message();

    this.threadsService.currentThread.subscribe(
      (thread: Thread) => {
        this.currentThread = thread;
      });

    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
```

```
        });

      this.messages
        .subscribe(
          (messages: Array<Message>) => {
            setTimeout(() => {
              this.scrollToBottom();
            });
        });
    }

    onEnter(event: any): void {
      this.sendMessage();
      event.preventDefault();
    }

    sendMessage(): void {
      let m: Message = this.draftMessage;
      m.author = this.currentUser;
      m.thread = this.currentThread;
      m.isRead = true;
      this.messagesService.addMessage(m);
      this.draftMessage = new Message();
    }

    scrollToBottom(): void {
      let scrollPane: any = this.el
        .nativeElement.querySelector('.msg-container-base');
      scrollPane.scrollTop = scrollPane.scrollHeight;
    }

  }
```

11.5 ChatMessage 组件

每条消息都是通过ChatMessage组件渲染的，如图11-5所示。

该组件相对简明，其主要逻辑是根据消息是否由当前用户所创建来渲染出略有不同的视图。如果该消息不是当前用户创建的，就认为消息是收到的（incoming）。

我们先从定义@Component开始。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  inputs: ['message'],
  selector: 'chat-message',
```

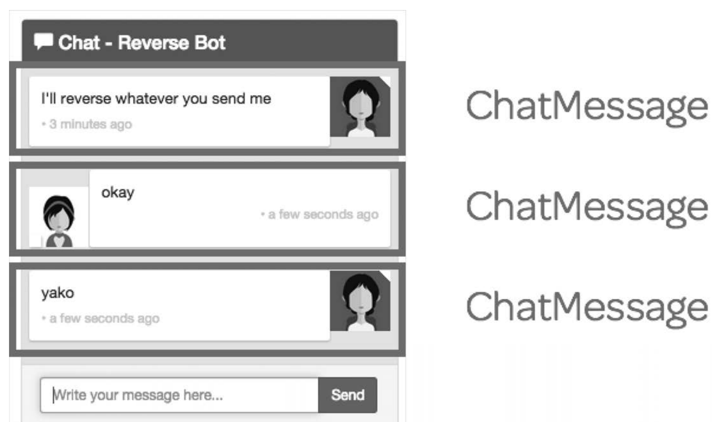


图11-5　ChatMessage组件

11.5.1　设置 incoming 属性

记住，每个ChatMessage组件都属于一条Message。因此，要在ngOnInit方法里订阅currentUser流并根据这条Message是否由当前用户所创建来设置incoming。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
export class ChatMessage implements OnInit {
  message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(private userService: UserService) {
  }

  ngOnInit(): void {
    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
        });
  }

}
```

11.5.2　ChatMessage 的 template

在template中有两处值得注意：

(1) FromNowPipe管道

(2) [ngClass]属性

先来看看它的代码。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
@Component({
  inputs: ['message'],
  selector: 'chat-message',
  template: `
  <div class="msg-container"
       [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

    <div class="avatar"
         *ngIf="!incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>

    <div class="messages"
      [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
      <p>{{message.text}}</p>
      <p class="time">{{message.author.name}} • {{message.sentAt | fromNow}}</p>
    </div>

    <div class="avatar"
         *ngIf="incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>
  </div>
  `
})
```

FromNowPipe是一个管道，把消息的发送时间转换为像“×秒前”这样对用户友好的信息。如你所见，我们要这样用它：{{message.sentAt | fromNow}}。



FromNowPipe使用优秀的moment.js[①]类库。如果你想学习如何创建自定义管道，可以阅读FromNowPipe的源代码：code/rxjs/chat/app/ts/util/FromNowPipe.ts。

我们也在视图中充分利用了ngClass。当这样写时：

```
[ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

我们是在告诉Angular，如果incoming为真就使用msg-receive类（否则使用msg-sent类）。

借助incoming属性，我们就能以不同的形式来显示收到和发出的消息。

① http://momentjs.com/

11.5.3 完整的 ChatMessage 代码清单

下面是完整的ChatMessage组件。

code/rxjs/chat/app/ts/components/ChatWindow.ts

```
import {
  Component,
  OnInit,
  ElementRef,
  ChangeDetectionStrategy
} from '@angular/core';
import {
  MessagesService,
  ThreadsService,
  UserService
} from '../services/services';
import {Observable} from 'rxjs';
import {User, Thread, Message} from '../models';

@Component({
  inputs: ['message'],
  selector: 'chat-message',
  template: `
  <div class="msg-container"
       [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

    <div class="avatar"
         *ngIf="!incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>

    <div class="messages"
      [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
      <p>{{message.text}}</p>
      <p class="time">{{message.author.name}} • {{message.sentAt | fromNow}}</p>
    </div>

    <div class="avatar"
         *ngIf="incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>
  </div>
  `
})
export class ChatMessage implements OnInit {
  message: Message;
  currentUser: User;
  incoming: boolean;

  constructor(private userService: UserService) {
  }
```

```
  ngOnInit(): void {
    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
          if (this.message.author && user) {
            this.incoming = this.message.author.id !== user.id;
          }
        });
  }

}

@Component({
  selector: 'chat-window',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">

            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container">
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>

            <div class="panel-body msg-container-base">
              <chat-message
                 *ngFor="let message of messages | async"
                 [message]="message">
              </chat-message>
            </div>

            <div class="panel-footer">
              <div class="input-group">
                <input type="text"
                       class="chat-input"
                       placeholder="Write your message here..."
                       (keydown.enter)="onEnter($event)"
                       [(ngModel)]="draftMessage.text" />
                <span class="input-group-btn">
                  <button class="btn-chat"
                    (click)="onEnter($event)"
                    >Send</button>
                </span>
              </div>
```

11

```
          </div>

        </div>
      </div>
    </div>
  </div>
  `
})
export class ChatWindow implements OnInit {
  messages: Observable<any>;
  currentThread: Thread;
  draftMessage: Message;
  currentUser: User;

  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService,
              private userService: UserService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    this.messages = this.threadsService.currentThreadMessages;

    this.draftMessage = new Message();

    this.threadsService.currentThread.subscribe(
      (thread: Thread) => {
        this.currentThread = thread;
      });

    this.userService.currentUser
      .subscribe(
        (user: User) => {
          this.currentUser = user;
        });

    this.messages
      .subscribe(
        (messages: Array<Message>) => {
          setTimeout(() => {
            this.scrollToBottom();
          });
        });
  }

  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }

  sendMessage(): void {
    let m: Message = this.draftMessage;
    m.author = this.currentUser;
    m.thread = this.currentThread;
```

```
    m.isRead = true;
    this.messagesService.addMessage(m);
    this.draftMessage = new Message();
  }

  scrollToBottom(): void {
    let scrollPane: any = this.el
      .nativeElement.querySelector('.msg-container-base');
    scrollPane.scrollTop = scrollPane.scrollHeight;
  }

}
```

11.6 ChatNavBar 组件

我们要讨论的最后一个组件是ChatNavBar。导航条中会显示当前用户的未读消息数，如图11-6所示。

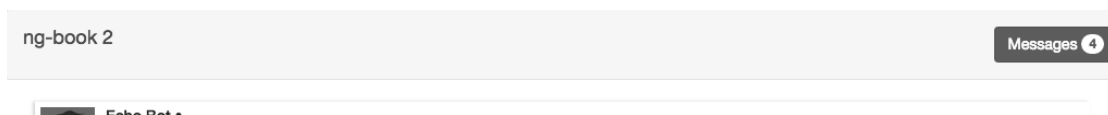


图11-6 ChatNavBar组件中的未读数



试验未读消息数量最好的办法是使用等待机器人（Waiting Bot）。如何你还没有试过，尝试发消息“3”给等待机器人，然后切换到其他聊天窗口。等待机器人会等3秒再给你回复消息，这样你就会看到未读消息数量的增长。

11.6.1 ChatNavBar 的@Component

首先，我们定义了非常简单的@Component配置。

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
@Component({
  selector: 'nav-bar',
```

11.6.2 ChatNavBar 控制器

ChatNavBar控制器唯一需要做的就是记录unreadMessagesCount属性。这其实比表面看上去稍微复杂一些。

最简明的方式就是监听messagesService.messages，然后计算属性isRead是false的Messages数量总和。对于当前会话外的所有消息，这种方法可以正常工作。然而，当messages

流发出新值时，无法保证当前会话的新消息被标记为已读。

最安全的方式就合并messages流和currentThread流，以确保不会把任何属于当前会话的消息算入总数。

我们用combineLatest操作符来进行实现（本章前面也使用过它）。

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
export class ChatNavBar implements OnInit {
  unreadMessagesCount: number;

  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _.reduce(
            messages,
            (sum: number, m: Message) => {
              let messageIsInCurrentThread: boolean = m.thread &&
                currentThread &&
                (currentThread.id === m.thread.id);
              if (m && !m.isRead && !messageIsInCurrentThread) {
                sum = sum + 1;
              }
                return sum;
            },
            0);
      });
  }
}
```

如果你不熟悉TypeScript的话，会觉得上面的语法有些不太容易理解。我们在combineLatest回调函数中返回了一个数组，这个数组包含两个元素：currentThread和messages。

然后我们订阅了combineLatest操作符返回的流，在函数调用中解构这些对象。接下来，我们用reduce化简了messages集合，对所有未读并且不属于当前会话的消息进行计数。

11.6.3　ChatNavBar 的 template

在视图中，唯一需要做的事就是显示unreadMessagesCount属性。

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
@Component({
  selector: 'nav-bar',
  template: `
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="https://ng-book.com/2">
          <img src="${require('images/logos/ng-book-2-minibook.png')}"/>
           ng-book 2
        </a>
      </div>
      <p class="navbar-text navbar-right">
        <button class="btn btn-primary" type="button">
          Messages <span class="badge">{{unreadMessagesCount}}</span>
        </button>
      </p>
    </div>
  </nav>
  `
```

11.6.4 完整的 ChatNavBar 组件

下面是完整的ChatNavBar组件代码清单。

code/rxjs/chat/app/ts/components/ChatNavBar.ts

```
import {Component, OnInit} from '@angular/core';
import {MessagesService, ThreadsService} from '../services/services';
import {Message, Thread} from '../models';
import * as _ from 'underscore';

@Component({
  selector: 'nav-bar',
  template: `
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
        <a class="navbar-brand" href="https://ng-book.com/2">
          <img src="${require('images/logos/ng-book-2-minibook.png')}"/>
           ng-book 2
        </a>
      </div>
      <p class="navbar-text navbar-right">
        <button class="btn btn-primary" type="button">
          Messages <span class="badge">{{unreadMessagesCount}}</span>
        </button>
      </p>
    </div>
  </nav>
  `
})
```

```
export class ChatNavBar implements OnInit {
  unreadMessagesCount: number;

  constructor(private messagesService: MessagesService,
              private threadsService: ThreadsService) {
  }

  ngOnInit(): void {
    this.messagesService.messages
      .combineLatest(
        this.threadsService.currentThread,
        (messages: Message[], currentThread: Thread) =>
          [currentThread, messages] )

      .subscribe(([currentThread, messages]: [Thread, Message[]]) => {
        this.unreadMessagesCount =
          _.reduce(
            messages,
            (sum: number, m: Message) => {
              let messageIsInCurrentThread: boolean = m.thread &&
                currentThread &&
                (currentThread.id === m.thread.id);
              if (m && !m.isRead && !messageIsInCurrentThread) {
                sum = sum + 1;
              }
              return sum;
            },
            0);
      });
  }
}
```

11.7　总结

好了，把它们全部放在一起，就是一个完整的聊天应用了（如图11-7所示）!

查看文件code/redux/angular2-redux-chat/app/ts/ChatExampleData.ts，你会发现我们已经写好了少量可以跟你聊天的机器人。下面是从反转机器人中截取的一些代码：

```
let rev: User = new User("Reverse Bot", require("images/avatars/female-avatar-4.png"));
let tRev: Thread = new Thread("tRev", rev.name, rev.avatarSrc);
```

code/rxjs/chat/app/ts/ChatExampleData.ts

```
    messagesService.messagesForThreadUser(tRev, rev)
      .forEach( (message: Message): void => {
        messagesService.addMessage(
          new Message({
            author: rev,
            text: message.text.split('').reverse().join(''),
            thread: tRev
          })
```

```
  );
},
```

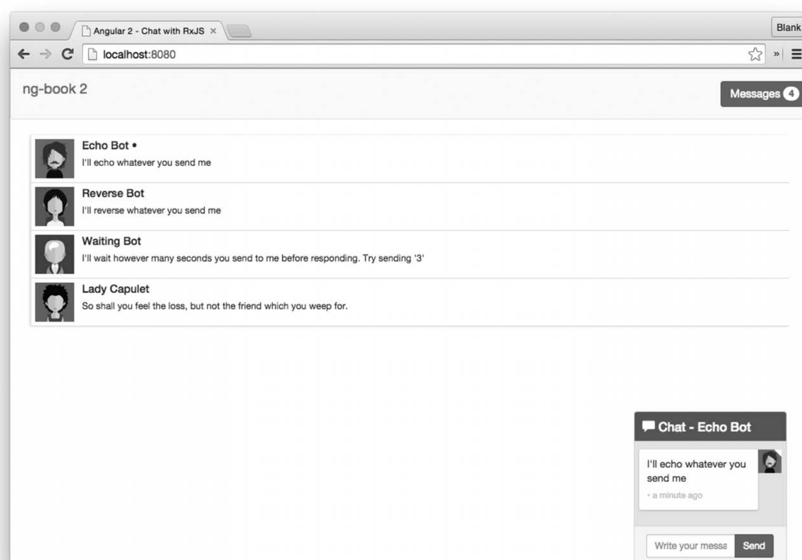


图11-7 完成后的聊天应用

如你所见，我们已经通过messagesForThreadUser方法为反转机器人订阅了消息。你可以试着写几个自己的机器人。

11.8 更进一步

改进这个聊天应用的一些方法包括加强RxJS的使用并连接到一个真实的API。发起API请求的方法我们已经在第6章中讨论过了。眼下请尽情享受你的聊天应用吧！

第12章

基于TypeScript的Redux简介

本章及下一章将着眼于一种叫作Redux的数据架构。本章将讨论Redux背后的理念，建造一个自己的迷你版Redux并把它连接到Angular。在下一章中，我们将使用Redux构建一个更大的应用。

到目前为止，我们的大多数项目都在通过一种相当直接的方式管理状态：从服务中获取数据，然后在组件中渲染数据。在组件树中，值是沿着自上而下的方向传递的。

对于比较小的应用来说，这种管理方式已经足够了；但随着应用的成长，让多个组件来管理状态的不同部分将变得难以处理。比如，通过组件树向下传递所有值的方式有如下缺点。

- *属性的间接传递*：为了让任何组件都可以获取到应用的状态，我们不得不通过inputs属性向下传递值。这意味着我们会借助很多中间组件来传递状态，而这些中间组件既不使用也不关心传递的状态。
- *重构不灵活*：传递inputs属性时要贯穿整个组件树，从而导致父子组件之间产生耦合，而这些耦合通常都是不必要的。这样，试图把一个子组件放入组件树的其他层级中会变得非常困难，因为我们必须修改所有新的父级组件来传递状态。
- *状态树和DOM树不匹配*：状态的"形状"往往和视图/组件层级的"形状"不匹配。当我们需要引用组件树一个较远分支中的数据时，通过组件树的属性来传递所有值就会使我们陷入困境。
- *应用中到处都是状态*：如果通过组件来管理状态，就很难获取应用整体状态的快照。因此很难知道哪个组件"拥有"一条特定的数据以及哪些组件关心该数据的变化。

把数据从组件中提取出来并放到服务中会有很大的帮助。至少，如果服务是数据的"拥有者"，那么对于把数据放在哪里，我们就有更清晰的概念。但这也带来了一个新问题：关于"让服务拥有数据"的最佳实践又是什么呢？有什么可以遵循的模式吗？当然有！

本章会讨论一种叫作Redux的数据架构模式，其设计初衷就是要解决这些问题。我们将自己实现一个Redux，它会把所有的状态都存储在一个地方。这种"把所有应用状态都存在同一个地

方”的想法乍听起来可能有点疯狂，但最终会给你惊喜。

12.1 Redux

如果你还没听说过Redux，可以到其官网http://redux.js.org/查看相关内容。网络应用的数据架构一直在进化，搭建数据架构的传统方式已经不能很好地适应大型网络应用。因为功能强大且易于理解，Redux如今非常流行。

数据架构是一个复杂的话题，而Redux的最大优点可能是它的简单性。如果把Redux剥离得只剩核心代码，其代码行数将不到100行。

通过把Redux用作应用的骨架，我们可以构建出更容易理解的富网络应用。首先，我们来看看如何编写一个迷你版Redux，稍后再把这些概念应用到一个更大的应用程序中，以更好地理解Redux的工作模式。



有人尝试使用Redux或新建一个受Redux启发的、能与Angular协同工作的系统。以下是两个著名的例子：

- ngrx/store[1]
- angular2-redux[2]

ngrx是一个受Redux启发的架构，也是可观察对象的重度使用者。angular2-redux则依赖于Redux并添加了一些Angular的辅助类（依赖注入、可观察对象包装）。

这里不会使用它们。为了在不引入新依赖的前提下更好地展示概念，我们将直接使用Redux。当然，在你编写自己的应用时，这两个类库可能会对你有所帮助。

Redux：核心概念

Redux的核心概念有：

- 应用的所有数据都放在一个叫作state的数据结构之中，而state存放在store中；
- 应用从store中读取state；
- store永远不会被直接修改；
- action描述发生了什么，由用户交互（和其他代码）触发；
- 通过调用一个叫作reducer的函数来结合旧的state和action会创建出新的state（如图12-1所示）。

① https://github.com/ngrx/store
② https://github.com/InfomediaLtd/angular2-redux

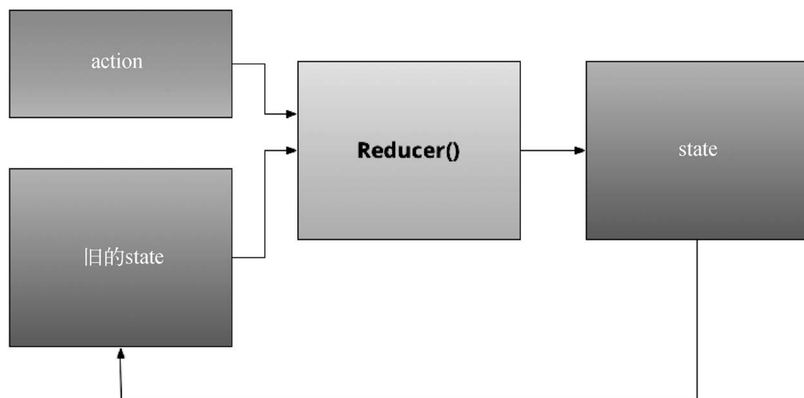


图12-1　Redux的内核

如果以上几点还不够清楚的话，也不用担心。本章的其余部分会把这些概念应用到实践中。

12.2　Redux 核心概念

12.2.1　reducer 是什么

我们先来讨论reducer（归集器）。reducer的概念是：接收旧的state和action并返回新的state。

reducer必须是一个纯函数[①]。也就是说：

(1) 它不能直接修改当前的state；

(2) 它不会使用参数之外的任何数据。

换句话说，一个纯函数在参数不变的情况下，总是会返回同一个值；而且纯函数不会调用任何会对外界产生影响的函数。比如，没有数据库调用，没有HTTP请求，也不会改变外部的数据结构。

reducer应始终把当前state当作只读的。reducer不应该改变state，而是应该返回一个新的state。（通常，新的state会从复制原有state开始，但我们不应该自己动手复制它。）

下面来定义我们的第一个reducer。记住，reducer涉及以下三点。

(1) `Action`：定义要做什么（能带可选参数）。

(2) `state`：存储应用中的所有数据。

(3) `Reducer`：接收`state`和`Action`并返回一个新的state。

① https://en.wikipedia.org/wiki/Pure_function

12.2.2 定义 Action 和 Reducer 的接口

因为我们使用TypeScript是为了确保全程都是带类型的，所以先为Action和Reducer设计一套接口。

1. Action接口

Action接口如下所示。

code/redux/angular2-redux-chat/minimal/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}
```

注意Action有两个字段：

(1) type

(2) payload

type是一个标识字符串，用来描述action的类型，比如INCREMENT或ADD_USER。payload可以是任意类型的对象。payload?中的?表示这个字段是可选的。

2. Reducer接口

Reducer接口如下所示。

code/redux/angular2-redux-chat/minimal/tutorial/01-identity-reducer.ts

```
interface Reducer<T> {
  (state: T, action: Action): T;
}
```

Reducer使用了TypeScript中一种名叫泛型的特性。在这个例子中，T就是state的类型。注意，这里我们要表达的是：有效的Reducer就是一个函数，它接收state（类型为T）和action并返回一个新的state（类型也是T）。

12.2.3 创建第一个 Reducer

最简单的reducer返回state本身。（可以把它叫作identity reducer，因为它在state上应用了"identity函数"[1]。这也是所有reducer的默认情况，我们很快就会看到。）

code/redux/angular2-redux-chat/minimal/tutorial/01-identity-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};
```

① https://en.wikipedia.org/wiki/Identity_function

注意，这个Reducer通过语法Reducer<number>把泛型中的类型固定为number。我们很快就会定义一些比数字更复杂的state。

我们还没有使用Action，但已经可以试用这个Reducer了。



运行本节的示例

你可以在code/redux文件夹中找到本章的代码。如果示例是可运行的，那么你就会在代码块上方看到文件名。

在本节中，这些例子是**在浏览器之外通过node.js来运行的**。因为这些例子中用的是TypeScript，所以你应该使用命令行工具ts-node（而不是直接使用node）来运行它们。

可以运行下面的命令来安装ts-node：

```
npm install -g ts-node
```

也可以在code/redux/angular2-redux-chat目录下运行npm install，然后调用./node_modules/.bin/ts-node --noProject。

比如，要运行上面的例子，你需要输入下列命令（不要输入$符）：

```
$ cd code/redux/angular2-redux-chat/minimal/tutorial
$ ../../node_modules/.bin/ts-node --noProject 01-identity-reducer.ts
```

在我们告诉你把运行环境切换到浏览器之前，本章其余的代码也都用同样的步骤运行。

12.2.4　运行第一个 Reducer

把所有代码整合起来并运行这个reducer。

code/redux/angular2-redux-chat/minimal/tutorial/01-identity-reducer.ts

```
interface Action {
  type: string;
  payload?: any;
}

interface Reducer<T> {
  (state: T, action: Action): T;
}

let reducer: Reducer<number> = (state: number, action: Action) => {
  return state;
};

console.log( reducer(0, null) ); // -> 0
```

运行下列命令：

```
$ cd code/redux/angular2-redux-chat/minimal/tutorial
$ ../../node_modules/.bin/ts-node --noProject 01-identity-reducer.ts
0
```

用这段代码作为示例似乎有点傻，但它教给了我们reducer的第一条原则：

> 默认情况下，reducer返回state本身。

在这个例子中，我们传入了一个值为数字0的state和一个值为null的action。reducer返回的结果是值为数字0的state。

但是我们还要做一些更有趣的事来改变state。

12.2.5　使用 action 调整计数器

我们最终的state会远比一个数字复杂得多。我们会把应用中的所有数据都保存在state中，这就需要为最终的state设计一种更好的数据结构。

不过，目前使用一个数字作为state可以让我们专注于其他问题。因此我们先沿用这种做法，state仅仅是一个用来存储计数器的数字。

假设我们希望改变state的数值。记住，我们不会在Redux中修改state。取而代之的是创建action，用来告诉reducer如何生成一个新的state。

让我们创建一个Action来改变计数器。要记住，Action唯一的必选属性就是type。我们可以像这样来定义第一个action：

```
let incrementAction: Action = { type: 'INCREMENT' }
```

我们还应该创建第二个action，它负责通知reducer让计数器变小：

```
let decrementAction: Action = { type: 'DECREMENT' }
```

现在有了这些action，我们来试试在reducer中使用它们。

code/redux/angular2-redux-chat/minimal/tutorial/02-adjusting-reducer.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  if (action.type === 'INCREMENT') {
    return state + 1;
  }
  if (action.type === 'DECREMENT') {
    return state - 1;
  }
  return state;
};
```

现在可以试用完整的reducer了。

code/redux/angular2-redux-chat/minimal/tutorial/02-adjusting-reducer.ts

```
let incrementAction: Action = { type: 'INCREMENT' };

console.log( reducer(0, incrementAction )); // -> 1
console.log( reducer(1, incrementAction )); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };

console.log( reducer(100, decrementAction )); // -> 99
```

漂亮！现在会根据传给reducer的action来决定返回的新state的值。

12.2.6　reducer 的 `switch`

我们通常把reducer的主体代码换成`switch`语句，而不是一大堆`if`。

code/redux/angular2-redux-chat/minimal/tutorial/03-adjusting-reducer-switch.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  default:
    return state; // <-- dont forget!
  }
};

let incrementAction: Action = { type: 'INCREMENT' };
console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2

let decrementAction: Action = { type: 'DECREMENT' };
console.log(reducer(100, decrementAction)); // -> 99

// any other action just returns the input state
let unknownAction: Action = { type: 'UNKNOWN' };
console.log(reducer(100, unknownAction)); // -> 100
```

注意`switch`语句的`default`分支要返回`state`本身。当传入一个未知的action时，这将确保程序不会报错而且我们能得到原始的`state`值。



问：等一下！难道要把应用中所有的state都放在一个庞大的switch语句中吗？
答：既是又不是。

如果这是你第一次接触Redux的reducer，那么“对应用中state的所有更改都是一个庞大switch语句的结果”可能会让你感到奇怪。你应该知道下面两点。

(1) 在一个地方集中管理state的变化对于维护程序有莫大的帮助，具体来说是因为当把所有状态都集中在一起时就很容易查出哪里发生了变化。（此外，你可以轻松地定位state的变化是哪个action的结果，因为你可以把action的type属性作为关键字在代码中进行搜索。）

(2) 你可以（而且经常会）将reduer分解成若干sub-reducer（子reducer），它们各自负责管理state树中的一个不同分支。我们稍后会进行讨论。

12.2.7 action 的“参数”

在上个例子中，我们的action只包含一个type属性，用来告诉reducer是递增还是递减这个state。

然而，应用的变化通常是无法通过单一的值来描述清楚的，而是需要一些参数来描述这种变化。这就是在Action里有payload字段的原因了。

在这个计数器示例中，如果我们想要让计数器增加9。一种做法是发送9次INCREMENT action，但这样做效率太低，尤其是在想增加一个较大数值的时候，如9000。

替代方案是增加一个PLUS action。它用payload参数来发送一个数字，这个数字表示计数器要增加的值。定义这个action很简单：

```
let plusSevenAction = { type: 'PLUS', payload: 7 };
```

接下来，要支持这个action，就要在reducer里添加一个新的case分支来处理PLUS action。

code/redux/angular2-redux-chat/minimal/tutorial/04-plus-action.ts

```
let reducer: Reducer<number> = (state: number, action: Action) => {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  case 'PLUS':
    return state + action.payload;
  default:
    return state;
  }
};
```

PLUS分支会把action.payload中的任何数字累加到state上。下面来试试看。

code/redux/angular2-redux-chat/minimal/tutorial/04-plus-action.ts

```
console.log( reducer(3, { type: 'PLUS', payload: 7}) ); // -> 10
console.log( reducer(3, { type: 'PLUS', payload: 9000}) ); // -> 9003
console.log( reducer(3, { type: 'PLUS', payload: -2}) ); // -> 1
```

我们在第一行接收的state是3，然后加上7，得到的结果是10。漂亮！不过，请注意当我们传递state的时候，它并没有真的发生变化。也就是说，我们没有保存reducer变化产生的结果，不能在之后的action中复用它。

12.3　保存 state

这些reducer都是纯函数，不会改变外部环境。问题在于，应用中的一切都在不断变化。特别是在state变化后，我们必须在某个地方保留这个新的state。

在Redux中，state是保存在store里的。store负责运行reducer然后保存新的state。我们来看一个最简单的store。

code/redux/angular2-redux-chat/minimal/tutorial/05-minimal-store.ts

```
class Store<T> {
  private _state: T;

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
  }
}
```

注意Store是泛型的，我们指定state的类型为泛型T，并用私有变量_state来存储state。

Store还应该有一个Reducer，它同样是泛型的，泛型的类型是T。这是因为每个store都和一个特定的reducer紧密相关。我们用私有变量reducer来存储这个Reducer。



在Redux中，每个应用通常只有一个store和一个顶层reducer。

让我们来仔细看看State中的每个方法：

- 在构造函数中把_state变量设置为初始的state；
- getState()直接返回当前的_state变量；
- dispatch接收一个action并把它传给reducer，然后用返回值来更新_state变量的值。

注意dispatch方法不返回任何值。它只更新store中的state（结果返回之后）。这是Redux中的一条重要原则：分发（dispatch）action是一种“触发并忘记”的策略。分发action并不直接操作state，所以它也不返回新的state。

当我们分发action的时候，会发送一个关于发生了什么的通知。如果想要了解系统的当前状态，就必须检查store中的state。

12.3.1 使用 store

我们来试试store。

code/redux/angular2-redux-chat/minimal/tutorial/05-minimal-store.ts

```
// create a new store
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 1

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // -> 2

store.dispatch({ type: 'DECREMENT' });
console.log(store.getState()); // -> 1
```

先创建一个新的Store对象并保存在store变量中。我们可以使用这个变量来获取当前的state并且分发action。

state初始值为0，然后进行两次INCREMENT、一次DECREMENT，最终的state值是1。

12.3.2 使用 subscribe 进行通知

Store记录着发生的变化，这很不错；但是在上个示例中，我们必须用store.getState()询问state的变化。如果一个新的action分发后能立刻让我们知道就好了，这样我们就能作出响应了。要做到这一点，可以实现观察者模式（observer pattern）。也就是说，我们会注册一个回调函数用来订阅所有的变化。

我们希望它这样工作：

(1) 我们用subscribe注册一个监听函数；

(2) 当dispatch被调用时，我们遍历所有的监听器并逐个调用它们，它们会负责通知大家这个state发生了变化。

1. 注册监听器

监听回调函数是没有参数的函数。我们来定义一个接口，以便于描述。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
interface ListenerCallback {
  (): void;
}
```

订阅一个监听器后，我们可能还需要取消订阅，因此也为取消订阅函数定义个接口。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
interface UnsubscribeCallback {
  (): void;
}
```

这段代码没什么内容，它只是另一个没有参数的函数，也没有返回值。但定义这些类型能让我们的代码更容易阅读。

store还要保存一个ListenerCallbacks的列表。我们把这个列表加到Store中。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];
```

接着我们就可以用subscribe函数把监听器添加到_listeners列表中了。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
```

subscribe接收一个ListenerCallback参数（也就是一个没有参数、没有返回值的函数）并返回UnsubscribeCallback（方法签名同上）。添加监听器很简单：只要用push方法把它追加到_listeners数组中就可以了。

它的返回值是一个函数。这个函数会修改_listeners列表，把刚加上的listener过滤掉。也就是说，它返回UnsubscribeCallback函数，调用此函数就会把刚加上的listener从列表中移除。

2. 通知监听器

每当state发生变化时，我们都要调用这些监听函数。也就是说，无论是分发了一个新的action

还是state发生变化，我们都要调用所有监听器。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }
```

3. 完整的store

稍后我们会亲自尝试这个store，不过现在先看看Store最新的完整代码清单。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
class Store<T> {
  private _state: T;
  private _listeners: ListenerCallback[] = [];

  constructor(
    private reducer: Reducer<T>,
    initialState: T
  ) {
    this._state = initialState;
  }

  getState(): T {
    return this._state;
  }

  dispatch(action: Action): void {
    this._state = this.reducer(this._state, action);
    this._listeners.forEach((listener: ListenerCallback) => listener());
  }

  subscribe(listener: ListenerCallback): UnsubscribeCallback {
    this._listeners.push(listener);
    return () => { // returns an "unsubscribe" function
      this._listeners = this._listeners.filter(l => l !== listener);
    };
  }
}
```

4. 试用`subscribe`

现在可以订阅这个store的变化了，试试看。

code/redux/angular2-redux-chat/minimal/tutorial/06-store-w-subscribe.ts

```
let store = new Store<number>(reducer, 0);
console.log(store.getState()); // -> 0

// subscribe
let unsubscribe = store.subscribe(() => {
  console.log('subscribed: ', store.getState());
```

```
});

store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 1
store.dispatch({ type: 'INCREMENT' }); // -> subscribed: 2

unsubscribe();
store.dispatch({ type: 'DECREMENT' }); // (nothing logged)

// decrement happened, even though we weren't listening for it
console.log(store.getState()); // -> 1
```

我们订阅了store并在其回调函数中输出日志`subscribed:`以及store的当前state。



注意，监听函数**并没有**把当前state作为参数传进来。尽管这个选择看起来有点奇怪，但这是因为还有另一些细节需要权衡。现在把state的**变更通知**和**当前state**分开会更利于思考。在此就不再深入探究了，要了解更多信息，请阅读https://github.com/reactjs/redux/issues/1707、https://github.com/reactjs/redux/issues/1513和https://github.com/reactjs/redux/issues/303。

我们保存了`unsubscribe`回调函数。接下来要注意，在调用`unsubscribe()`之后就不会再输出日志了。我们仍然可以分发action，但却看不到它的结果了，除非直接向store询问。



如果喜欢RxJS和可观察对象，你可能会想到，其实也可以用RxJS实现自己的订阅监听器。你可以重写`Store`，用可观察对象代替我们自行实现的订阅机制。

英雄所见略同。事实上，我们已经替你做好了，你可以在文件code/redux/angular2-redux-chat/minimal/tutorial/06b-rx-store.ts中找到示例代码。

如果你愿意使用RxJS作为应用的数据骨架，那么用RxJS实现`Store`就是一种有趣而强大的模式。

我们在这里并没有过多使用可观察对象，主要是因为我们想讨论Redux本身以及如何使用一个单独的state树来思考数据架构。Redux本身已经强大到不必借助可观察对象就可以在应用中使用了。

一旦你领悟了如何使用"正统"Redux，那么再加入可观察对象就一点也不难了（前提是你已经理解了RxJS）。我们先暂且使用"正统"Redux，本章结尾处会给出一些指引，告诉你如何使用基于可观察对象的Redux包装器。

12.3.3　Redux 核心

上面这个store就是Redux的基本内核。reducer接收当前state和action并返回一个新的state，这个state会保存在store中。

想要构建一个用于生产环境的大型网络应用，我们显然还要添加更多。但是，我们稍后涉及

的所有新概念都将以这样一个简单的概念为基础：state是不可改变的，是集中存储的。如果掌握了之前提到的这些概念，也可以发明一些能用在高级Redxu应用中的模式（以及类库）。

在Redux的日常使用过程中，还有许多我们未曾涉及的方面。比如，我们需要知道：

- 如何在state中精心处理更复杂的数据结构；
- 当state发生变化时，如何不必轮询state就得到通知（使用订阅）；
- 如何拦截分发进行调试（也叫middleware）；
- 如何计算派生值（使用选择器）；
- 如何把一个大型reducer分解成许多可维护的小型reducer（并重新组合）；
- 如何处理异步数据。

我们将在本章的剩余部分和下一章中逐一解释这些问题并讲解常用的模式。

我们首先介绍如何在state中处理更复杂的数据结构。为此，我们需要一个比计数器更有意思的示例。那就构建一个聊天应用吧，用户可以用它向彼此发送消息。

12.4　消息应用

在这个聊天应用中（以及所有Redux应用中）数据模型有三个主要部分：

(1) state

(2) action

(3) reducer

12.4.1　消息应用的 state

计数器应用中的state只是一个数字，而在这个消息应用中，state是一个对象。

这个state对象只有一个属性messages。messages是一个字符串数组，每个字符串表示应用中的一条消息。例如：

```
// an example `state` value
{
  messages: [
    'here is message one',
    'here is message two'
  ]
}
```

我们可以这样定义该应用中的state类型。

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
interface AppState {
  messages: string[];
}
```

12.4.2　消息应用的 action

这个应用将处理两个action：ADD_MESSAGE和DELETE_MESSAGE。

action对象ADD_MESSAGE永远都有一个属性message，这个属性表示添加到state中的消息。action对象ADD_MESSAGE的模型如下：

```
{
  type: 'ADD_MESSAGE',
  message: 'Whatever message we want here'
}
```

action对象DELETE_MESSAGE会从state中删除一条指定的消息。这里的问题在于，我们要指出想删除的是哪条消息。

如果消息的数据结构是对象的话，可以在每条消息创建的时候赋予它一个id属性。然而，为了让这个示例尽可能简单，消息只是单纯的字符串，因此我们只能用另一种方式来删除消息了。目前最简单的方式就是直接使用消息数组里的索引（可以看作事实性的ID）。

明白这一点之后，action对象DELETE_MESSAGE的模型如下：

```
{
  type: 'DELETE_MESSAGE',
  index: 2                // <- or whatever index is appropriate
}
```

我们可以用TypeScript的语法interface ... extends来定义这些action的类型。

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
interface AddMessageAction extends Action {
  message: string;
}

interface DeleteMessageAction extends Action {
  index: number;
}
```

这样AddMessageAction就能指定一条消息了，而DeleteMessageAction也可以指定一个索引。

12.4.3　消息应用的 reducer

记住reducer需要处理两个action：ADD_MESSAGE和DELETE_MESSAGE。下面来分别讨论它们：

1. 处理ADD_MESSAGE

首先针对action.type使用switch语句并处理ADD_MESSAGE分支。

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
let reducer: Reducer<AppState> =
  (state: AppState, action: Action): AppState => {
  switch (action.type) {
  case 'ADD_MESSAGE':
    return {
      messages: state.messages.concat(
        (<AddMessageAction>action).message
      ),
    };
```



TypeScript的对象本身已经有类型了，为什么还要添加一个type字段呢？

要处理这种“多态分发”（polymorphic dispatch），有很多方式可供选择。想区分不同类型的action并在同一个reducer里处理它们，一种非常简明的方式是在type字段里存一个字符串（这里type的意思是“action的类型”）。从某种程度上说，你确实不必为每个action创建一个新的接口。

不过，用反射来实现对具体类型的switch会更令人满意。虽然类型守卫[1]开启了这种可能性，但当前版本的TypeScript还做不到这一点。

从广义上来说，类型只是一个编译阶段的概念。代码编译成JavaScript后，会丢失一些类型的元数据。

当然，如果你觉得对type字段进行switch很麻烦，希望直接使用语言特性来实现的话，也可以使用“装饰器反射元数据”技术[2]。目前，用一个简单的type字段就足够了。

2. 添加一项而不改变原有数据

当处理ADD_MESSAGE时，我们需要把给定的消息添加到state中。像所有的reducer一样，我们需要返回一个新的state。要记住，reducer必须是纯函数并且不会改变旧的state。

下面的代码有什么问题？

```
case 'ADD_MESSAGE':
  state.messages.push( action.message );
  return { messages: messages };
  // ...
```

问题在于这段代码改变了state.messages数组，也就是改变了旧的state。正确的做法是创建一个state.messages数组的副本并把新消息添加到这个副本中。

① https://basarat.gitbooks.io/typescript/content/docs/types/typeGuard.html
② http://blog.wolksoftware.com/decorators-metadata-reflection-in-typescript-from-novice-to-expert-part-4

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
case 'ADD_MESSAGE':
  return {
    messages: state.messages.concat(
      (<AddMessageAction>action).message
    ),
  };
```



语法<AddMessageAction>action会把action转换成更具体的类型。也就是说，reducer接收的是更通用的类型Action，它并没有messsage字段。如果这里我们没有进行转换，那么编译器就会报告说Action没有messsage字段。

但是，我们确实知道有一个ADD_MESSAGE action，所以就把它转化成一个AddMessageAction。使用圆括号来确保编译器知道我们要转化的是action而不是action.message。

记住，reducer必须返回一个新的AppState。当我们从reducer返回一个对象的时候，它必须匹配AppState的格式。在这个例子中，我们只需要一个关键字段messages；但在更复杂的state中，就要考虑更多字段了。

3. 删除一项而不改变原有数据

记住，当处理DELETE_MESSAGE action时，我们传入数组中消息的索引作为代理ID（另一种常见的做法是传入一个真实条目的ID）。另外，因为我们不想改变旧的messages数组，所以需要小心处理。

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
case 'DELETE_MESSAGE':
  let idx = (<DeleteMessageAction>action).index;
  return {
    messages: [
      ...state.messages.slice(0, idx),
      ...state.messages.slice(idx + 1, state.messages.length)
    ]
```

这里使用了两次slice操作符。首先获取要删除条目之前的所有条目，然后连接上其后的所有条目。



有4种不改变原有数据的常见操作：

- 往数组中添加一项；
- 从数组中移除一项；
- 添加或修改对象中的键；
- 从对象中移除键。

前两个（数组的）操作我们已经介绍过了。接下来我们将讨论更多关于对象的操作。目前需要知道的是一种使用Object.assign的常用方法，如下所示：

```
Object.assign({}, oldObject, newObject)
            // <-------<-------------
```

你可以认为Object.assign方法是从右至左地合并对象。newObject合并到oldObject，再合并到{}。这样，oldObject的所有字段都会保留，除非字段在newObject中也存在。无论是oldObject还是newObject都不会被改变。

当然，进行这些处理时要小心谨慎，因为很容易犯错。这也是很多人使用Immutable.js①的一个原因，Immutable.js是一组有助于加强不变性的数据结构。

12.4.4　试用 action

现在来尝试运行action。

code/redux/angular2-redux-chat/minimal/tutorial/07-messages-reducer.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Would you say the fringe was made of silk?'
} as AddMessageAction);

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Wouldnt have no other kind but silk'
} as AddMessageAction);

store.dispatch({
  type: 'ADD_MESSAGE',
  message: 'Has it really got a team of snow white horses?'
} as AddMessageAction);

console.log(store.getState());
// ->
```


① https://facebook.github.io/immutable-js/

```
// { messages:
// [ 'Would you say the fringe was made of silk?',
//   'Wouldnt have no other kind but silk',
//   'Has it really got a team of snow white horses?' ] }
```

我们先创建了一个新的store，然后调用store.getState()，从而看到一个空的messages数组。

接下来，我们往store中添加三条消息①。对于每条消息，我们都把type设为ADD_MESSAGE并把每个对象转换成AddMessageAction。

最后，我们把新的state打印出来，就能看到messages数组包含了所有这三条消息。

这三个dispatch语句都不够优雅，原因有以下两点。

(1) 每次都需要手动指定type字符串。我们也可以改用常量，但是如果什么都不用做就更好了。

(2) 需要手动转换成AddMessageAction。

我们应该创建一个函数来创建这些对象，而不是直接创建。编写函数来创建action的思想在Redux中很常见，因此这种模式有个名字：action creator。

12.4.5　action creator

我们要创建一个函数来创建ADD_MESSAGE action，而不是直接使用对象。

code/redux/angular2-redux-chat/minimal/tutorial/08-action-creators.ts

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message
    };
  }
  static deleteMessage(index: number): DeleteMessageAction {
    return {
      type: 'DELETE_MESSAGE',
      index: index
    };
  }
}
```

这里创建了一个类，它有两个静态方法addMessage和deleteMessage，分别返回AddMessageAction和DeleteMessageAction。

① https://en.wikipedia.org/wiki/The_Surrey_with_the_Fringe_on_Top



你不一定要用静态方法作为action creator，也可以使用普通的函数，命名空间中的函数，甚至是一个对象的实例方法等。关键是要用统一的方式来组织它们，让它们便于使用。

现在我们就改用新的action creator了。

code/redux/angular2-redux-chat/minimal/tutorial/08-action-creators.ts

```
let store = new Store<AppState>(reducer, { messages: [] });
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

console.log(store.getState());
// ->
// { messages:
// [ 'Would you say the fringe was made of silk?',
//   'Wouldnt have no other kind but silk',
//   'Has it really got a team of snow white horses?' ] }
```

这样感觉好多了！

它还有一个额外的好处：如果最终决定要改变消息的格式，我们不用更新任何一处`dispatch`语句。比如，假设我们要给每条消息增加创建时间，就可以在`addMessage`方法中添加一个`created_at`字段，那么现在所有的`AddMessageActions`都会有`created_at`字段：

```
class MessageActions {
  static addMessage(message: string): AddMessageAction {
    return {
      type: 'ADD_MESSAGE',
      message: message,
      // something like this
      created_at: new Date()
    };
  }
  // ....
```

12.4.6　使用真正的 Redux

现在我们已经写好了自己的迷你版Redux。你可能会问："要想使用真正的Redux还需要做什么？"谢天谢地，没有多少要做的。让我们更新一下代码，现在就改用真正的Redux。



如果你还没准备好，那就需要在code/redux/angular2-redux-chat/minimal/tutorial目录下运行命令npm install。

首先要做的是从redux包中导入Action、Reducer和Store。同时还导入了一个辅助函数createStore。

code/redux/angular2-redux-chat/minimal/tutorial/09-real-redux.ts

```
import {
  Action,
  Reducer,
  Store,
  createStore
} from 'redux';
```

接下来，让reducer创建初始的state，而不是在创建store的时候指定。这里，我们让reducer的默认参数来做这件事。采用这种方式，如果没有state传入（例如在初始化阶段中reducer被首次调用）就会使用初始的state。

code/redux/angular2-redux-chat/minimal/tutorial/09-real-redux.ts

```
let initialState: AppState = { messages: [] };

let reducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action) => {
```

reducer的其余部分都不用动，干得漂亮！

最后要做的是使用Redux的辅助函数createStore来创建store。

code/redux/angular2-redux-chat/minimal/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
```

之后一切正常！

code/redux/angular2-redux-chat/minimal/tutorial/09-real-redux.ts

```
let store: Store<AppState> = createStore<AppState>(reducer);
console.log(store.getState()); // -> { messages: [] }

store.dispatch(
  MessageActions.addMessage('Would you say the fringe was made of silk?'));

store.dispatch(
  MessageActions.addMessage('Wouldnt have no other kind but silk'));

store.dispatch(
  MessageActions.addMessage('Has it really got a team of snow white horses?'));

console.log(store.getState());
// ->
```

```
// { messages:
// [ 'Would you say the fringe was made of silk?',
//   'Wouldnt have no other kind but silk',
//   'Has it really got a team of snow white horses?' ] }
```

现在我们只是单纯地使用Redux来解决问题，下一步还要把Redux和我们的网络应用联系起来。开始行动吧。

12.5　在 Angular 中使用 Redux

在上一节中，我们学习了Redux的核心并展示了如何在Redux中创建reducer以及使用store管理数据。现在我们要更进一步，把Redux和Angular组件结合起来。

我们将在本节中创建一个最小化的Angular应用。该应用只有一个计数器，可以通过按钮来增加或减少计数（如图12-2所示）。

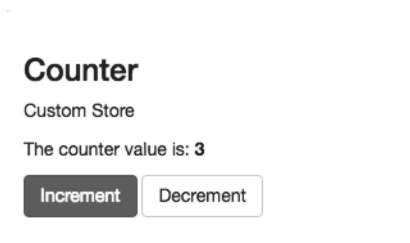


图12-2　计数器应用

这种小应用可以让我们专注于Redux和Angular之间的集成点。在下一节中，我们将进一步讨论更大的应用。目前，我们先来看看如何构建这个计数器应用！



我们没有在Redux和Angular之间使用任何辅助类库，而是直接集成它们。其实有很多开源类库可以简化这一过程，参见12.13节。
不过，一旦你理解了其背后的原理，使用这些类库也会容易得多。这里我们所做的一切都是为了让你更好地理解Redux背后的原理。

12.6　规划应用

你应该还记得，规划Redux应用的三个步骤是：

(1) 定义应用中央state的数据结构；

(2) 定义用来改变state的action；

(3) 定义一个reducer，用于接收旧的state和一个action并返回新的state。

对于这个应用来说，我们只是要增加或者减少计数。这个功能已经在上一节实现了，所以你会对本节的action、store和reducer感到非常熟悉。

我们要做的另外一件事就是，在编写Angular应用时决定在哪里创建组件。在这个应用中，有一个顶层组件CounterApp，它包含一个CounterComponent组件。CounterComponent组件则包含屏幕截图所示的那个视图。

大致上，我们要做以下几件事：

(1) 创建Store并通过依赖注入使它可以在整个应用中被访问到；

(2) 订阅Store的变化并在组件中显示出来；

(3) 当发生某些变化时（例如按下按钮时），我们将通过Store来分发一个action。

计划得差不多了，下面来看看如何在实践中应用！

12.7 组建 Redux

首先导入一些稍后要用到的东西。

code/redux/angular2-redux-chat/minimal/app.ts

```
import {
  Component
} from '@angular/core';
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
import {
  createStore,
  Store,
  StoreEnhancer
} from 'redux';
import { counterReducer } from './counter-reducer';
```

我们导入了Store（类）和createStore（辅助函数），之前用到过它们。我们还导入了一个叫作StoreEnhancer的新类，很快就会讲到它。

我们还从counter-reducer.ts中导入了reducer，从app-state.ts中导入了state的接口AppState。

12.7.1 定义应用的 state

让我们来看看AppState。

code/redux/angular2-redux-chat/minimal/app-state.ts

```
export interface AppState {
```

```
  counter: number;
};
```

这里把中央state的结构定义成了AppState，它是一个对象并且只有一个键counter（类型为number）。在下个示例（聊天应用）中，我们将讨论如何使用更复杂的state，但目前这样就足够了。

12.7.2 定义 reducer

接下来定义reducer，它负责处理应用state中计数器的增加和减少。

code/redux/angular2-redux-chat/minimal/counter-reducer.ts

```
mport {
  INCREMENT,
  DECREMENT
} from './counter-action-creators';

let initialState: AppState = { counter: 0 };

// Create our reducer that will handle changes to the state
export const counterReducer: Reducer<AppState> =
  (state: AppState = initialState, action: Action): AppState => {
    switch (action.type) {
    case INCREMENT:
      return Object.assign({}, state, { counter: state.counter + 1 });
    case DECREMENT:
      return Object.assign({}, state, { counter: state.counter - 1 });
    default:
      return state;
    }
  };
```

我们先导入了两个常量INCREMENT和DECREMENT，它们是由action creator导出的。虽然它们只是被简单地定义成了字符串'INCREMENT'和'DECREMENT'，但不错的是我们可以从编译器那里获得额外的帮助，以防打错字。我们稍后再来看看这些action creator。

initialState是一个AppState，它的counter属性为0。

counterReducer处理两个action：使当前计数器加1的INCREMENT以及使计数器减1的DECREMENT。这两个action都使用Object.assign来确保不会改变旧的state，而是创建一个新对象并把它作为新的state返回。

既然说到了这里，我们就来看看action creator。

12.7.3 定义 action creator

action creator是函数，返回的是定义action的对象。下面的increment和decrement函数会返回一个定义了合适type的对象。

code/redux/angular2-redux-chat/minimal/counter-action-creators.ts

```
import {
  Action,
  ActionCreator
} from 'redux';

export const INCREMENT: string = 'INCREMENT';
export const increment: ActionCreator<Action> = () => ({
  type: INCREMENT
});

export const DECREMENT: string = 'DECREMENT';
export const decrement: ActionCreator<Action> = () => ({
  type: DECREMENT
});
```

注意，action creator函数返回的是类型`ActionCreator<Action>`。`ActionCreator`是一个Redux定义的泛型类，可以用来定义action的创建函数。在这个例子中，我们使用的具体类是`Action`，但也可以使用一个更具体的类，比如上一节定义的`AddMessageAction`。

12.7.4　创建 store

现在有了reducer和state，我们可以这样创建store。

```
let store: Store<AppState> = createStore<AppState>(counterReducer);
```

不过，Redux有一点非常棒，那就是它有一组健壮的开发者工具（如图12-3所示）。特别是Chrome插件[1]，我们可以用它监控应用中的state以及分发action。

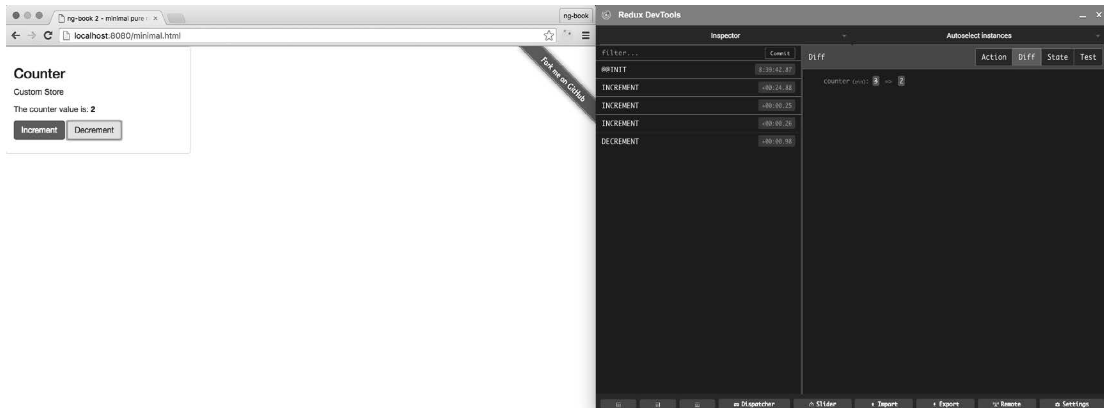


图12-3　带有Redux开发工具的计数器应用

① https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en

Redux DevTools最棒的一点是，它可以让我们清楚地观察到每个action如何流经本系统以及它对state的影响。



现在就去安装Redux DevTools中的Chrome插件吧！

要想使用开发者工具，我们必须先做一件事：把它添加到store中。

code/redux/angular2-redux-chat/minimal/app.ts

```
let devtools: StoreEnhancer<AppState> =
  window['devToolsExtension'] ?
  window['devToolsExtension']() : f => f;
```

并不是每个使用我们应用的人都安装好了Redux DevTools。上述代码会检查由Redux DevTools定义的`window.devToolsExtension`。如果它存在，我们就使用它；否则返回一个identity function（`f => f`），它会直接返回传给它的一切。



middleware是一个术语，表示用来强化另一个类库功能的函数。Redux DevTools是众多Redux middleware类库中的一个。Redux支持许多有趣的middleware，如果想自己写也很容易。
你可以在http://redux.js.org/docs/advanced/Middleware.html读到关于Redux middleware的更多内容。

为了使用这个`devtools`，我们把它当作middleware传给Redux的store。

code/redux/angular2-redux-chat/minimal/app.ts

```
let store: Store<AppState> = createStore<AppState>(
  counterReducer,
  devtools
);
```

现在，无论我们分发action还是改变state，都可以在浏览器中监测到了。

12.8　CounterApp 组件

现在已经设置好了Redux的内核，我们把注意力转向Angular组件。先来创建应用的顶层组件`CounterApp`。它将被用来引导（`bootstrap`）Angular。

code/redux/angular2-redux-chat/minimal/app.ts

```
@Component({
  selector: 'minimal-redux-app',
  template: `
  <div>
```

```
    <counter-component>
    </counter-component>
  </div>
  `
})
class CounterApp {
}
```

这个组件所做的一切就是创建CounterComponent的一个实例，我们马上就会定义它。在此之前，让我们先来启动应用。

12.9　提供 store

我们将用CounterApp作为应用的根组件。记住，由于这是一个Redux应用，我们需要让store的实例在应用的任何地方都能被访问到。该怎么做呢？我们将使用依赖注入技术。

还记得第8章提到过的吗？当希望通过依赖注入来获取某样东西时，我们就会在NgModule中使用providers配置项将其添加到providers列表中。

如果我们要把某样东西提供给依赖注入系统，需要指出两点：

(1) 用于指代这个可注入依赖的令牌；

(2) 注入依赖的方式。

通常，如果我们想提供一个单例服务，可能会像这样使用useClass选项：

```
{ provide: SpotifyService, useClass: SpotifyService }
```

在这个例子中，我们使用SpotifyService类作为依赖注入系统中的令牌。useClass选项会告诉Angular创建SpotifyService的一个实例，并且无论何时要求注入SpotifyService都会复用这个实例（也就是维护一个单例）。

不过使用这种方式有一个问题：我们不想让Angular创建store，因为之前已经用createStore创建好了。我们只想使用已创建好的store。

要这么做，就要使用provide中的useValue选项。之前我们已经使用过像API_URL这样的可配置值了：

```
{ provide: API_URL, useValue: 'http://localhost/api' }
```

还有一件事没有解决，那就是使用什么样的令牌来注入。store的类型是Store<AppState>。

code/redux/angular2-redux-chat/minimal/app.ts

```
let store: Store<AppState> = createStore<AppState>(
  counterReducer,
  devtools
);
```

Store并非一个类，而是一个接口。很不幸，我们不能使用接口作为依赖注入的键。



你也许想知道接口**为什么**不能作为依赖注入的键。答案就是，因为TypeScript的接口在编译完成后就会被移除，所以在运行环境中是获取不到的。
如果你想了解更多，请参见http://stackoverflow.com/questions/32254952/binding-a-class-to-an-interface、https://github.com/angular/angular/issues/135和http://victor-savkin. com/post/126514197956/dependency-injection-in-angular-1-and-angular-2。

这就表示我们需要创建自己的令牌，用来注入store。谢天谢地，Angular让这项任务变得很容易。我们在store的文件中创建这个令牌，这样就可以在应用的任何地方导入它。

code/redux/angular2-redux-chat/minimal/app-store.ts

```
import { OpaqueToken } from '@angular/core';

export const AppStore = new OpaqueToken('App.store');
```

这里创建了一个`const AppStore`，它使用Angular提供的`OpaqueToken`类。相对于直接注入字符串，`OpaqueToken`是一个更好的选择，因为它有助于避免命名冲突。

现在我们可以在`provide`中使用`AppStore`这个令牌了。开工！

12.10　启动应用

回到app.ts文件，我们创建一个`NgModule`来启动应用。

code/redux/angular2-redux-chat/minimal/app.ts

```
@NgModule({
  declarations: [
    CounterApp,
    CounterComponent
  ],
  imports: [ BrowserModule ],
  bootstrap: [ CounterApp ],
  providers: [
    {provide: AppStore, useValue: store }
  ]

})
class CounterAppAppModule {}

platformBrowserDynamic().bootstrapModule(CounterAppAppModule)
```

现在我们就可以通过注入`AppStore`在应用的任何地方引用Redux的store了。目前最需要它的地方就是`CounterComponent`。

12.11　CounterComponent

随着设置的完成，我们可以开始创建组件了。它实际上负责向用户显示计数器并提供按钮来让用户改变state。

12.11.1　import

我们先来看看导入。

code/redux/angular2-redux-chat/minimal/CounterComponent.ts

```
import {
  Component,
  Inject
} from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app-store';
import { AppState } from './app-state';
import * as CounterActions from './counter-action-creators';
```

我们从Redux中导入了`Store`以及我们自己的注入令牌`AppStore`，它可以让我们引用到store的单例。我们还导入了`AppState`类型，这有助于我们掌握中央state的结构。

最后，我们通过`* as CounterActions`语法导入了所有的action creator。这个语法会让我们调用`CounterActions.increment()`来创建一个`INCREMENT` action。

12.11.2　模板

我们来看看`CounterComponent`的模板（如图12-4所示）。

code/redux/angular2-redux-chat/minimal/CounterComponent.ts

```
@Component({
  selector: 'counter-component',
  template: `
    <div class="row">
      <div class="col-sm-6 col-md-4">
        <div class="thumbnail">
          <div class="caption">
            <h3>Counter</h3>
            <p>Custom Store</p>

            <p>
              The counter value is:
              <b>{{ counter }}</b>
            </p>

            <p>
              <button (click)="increment()"
```

```
                    class="btn btn-primary">
              Increment
            </button>
            <button (click)="decrement()"
                    class="btn btn-default">
              Decrement
            </button>
          </p>
        </div>
      </div>
    </div>
  </div>
  `
```

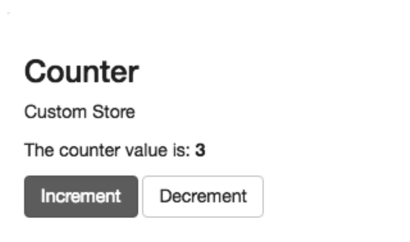


图12-4 计数器应用的模板

这里有三点需要注意：

(1) {{ counter }}用来显示计数器的值；

(2) 点击一个按钮时会调用increment()；

(3) 点击另一个按钮时会调用decrement()。

12.11.3 constructor

因为这个组件依赖于Store，所以我们要在构造函数中把它注入进来。这里示范的是我们如何使用自定义的AppStore令牌来注入依赖。

code/redux/angular2-redux-chat/minimal/CounterComponent.ts

```
export default class CounterComponent {
  counter: number;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.readState());
    this.readState();
  }

  readState() {
    let state: AppState = this.store.getState() as AppState;
    this.counter = state.counter;
```

```
  }

  increment() {
    this.store.dispatch(CounterActions.increment());
  }

  decrement() {
    this.store.dispatch(CounterActions.decrement());
  }
}
```

我们使用@Inject注解来注入AppStore。注意，我们把变量store的类型定义成了Store<AppState>。这里使用的注入令牌和用类作为注入令牌时（Angular能推断出要注入的是什么）略有不同。

我们把store设置为一个实例变量（使用private store）。有了store，我们就可以监听它的变化了。这里调用了store.subscribe和this.readState()；下面会定义readState。

只有当一个新的action被分发时，store才会调用subscribe，因此在这里需要确保至少手动调用readState一次，以保证组件可以获取到初始数据。

readState方法从store中读取state并把this.counter更新成最新值。因为this.counter是类的一个属性并在视图中绑定，所以Angular会检测到它发生了变化并重新渲染组件。

我们定义了两个辅助方法increment和decrement，它们分别把各自的action分发到store中。

12.11.4　整合

下面是CounterComponent的完整代码清单。

code/redux/angular2-redux-chat/minimal/CounterComponent.ts

```
import {
  Component,
  Inject
} from '@angular/core';
import { Store } from 'redux';
import { AppStore } from './app-store';
import { AppState } from './app-state';
import * as CounterActions from './counter-action-creators';

@Component({
  selector: 'counter-component',
  template: `
    <div class="row">
      <div class="col-sm-6 col-md-4">
        <div class="thumbnail">
          <div class="caption">
            <h3>Counter</h3>
            <p>Custom Store</p>
```

```
            <p>
              The counter value is:
              <b>{{ counter }}</b>
            </p>

            <p>
              <button (click)="increment()"
                      class="btn btn-primary">
                Increment
              </button>
              <button (click)="decrement()"
                      class="btn btn-default">
                Decrement
              </button>
            </p>
          </div>
        </div>
      </div>
    </div>
  `
})
export default class CounterComponent {
  counter: number;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.readState());
    this.readState();
  }

  readState() {
    let state: AppState = this.store.getState() as AppState;
    this.counter = state.counter;
  }

  increment() {
    this.store.dispatch(CounterActions.increment());
  }

  decrement() {
    this.store.dispatch(CounterActions.decrement());
  }
}
```

试一下（如图12-5所示）!

```
cd code/redux/angular2-redux-chat
npm install
npm run go
open http://localhost:8080/minimal.html
```

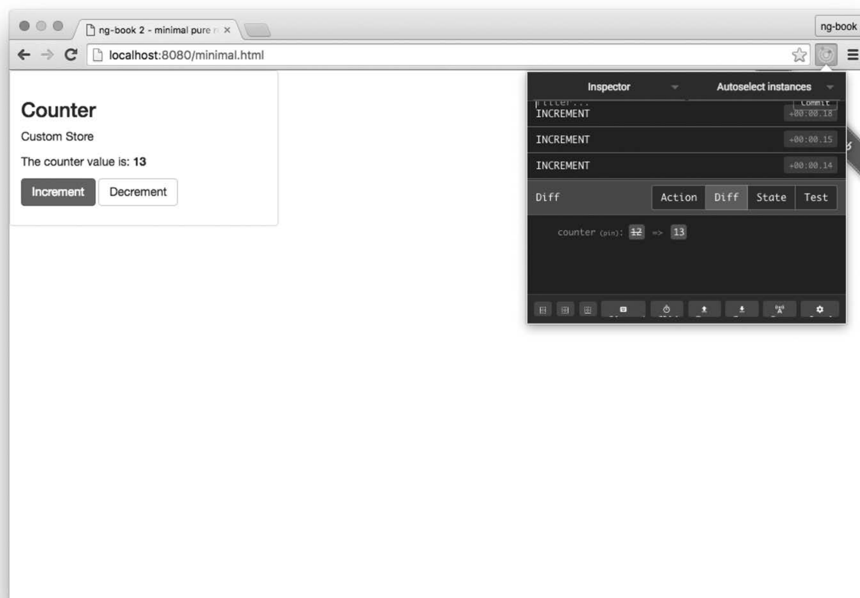


图12-5　工作中的计数器应用

恭喜！你已经创建了第一个Angular和Redux应用！

12.12　更进一步

现在我们已经使用Redux和Angular构建了一个基本的应用，还应该尝试构建一个更复杂的应用。当试图构建更大型的应用时，我们会遭遇新的挑战。

- 如何组合使用reducer?
- 如何从state的不同分支中提取数据?
- 如何组织Redux代码?

在下一章中，我们将构建一个聊天应用，并在其中处理所有这些问题！

12.13　参考资源

如果你想学习更多关于Redux的知识，下面是一些很不错的资源。

- Redux官网：http://redux.js.org/
- Redux作者的视频教程：https://egghead.io/courses/getting-started-with-redux

- 真实世界中的Redux（幻灯片展示）：https://speakerdeck.com/chrisui/real-world-redux
- 强大的高阶reducer：http://slides.com/omnidan/hor

要学习更多如何结合使用Redux和Angular内容，请查阅以下资源。

- angular2-redux：https://github.com/InfomediaLtd/angular2-redux
- ng2-redux：https://github.com/angular-redux/ng2-redux
- ngrx/store：https://github.com/ngrx/store

继续前进吧！

第 13 章

在Angular中引入Redux

Redux是一种流行且优雅的数据架构，我们在上一章学习了它的相关知识。我们还构建了一个非常基础的应用，结合了Angular组件和Redux的store。

在本章中，我们将进一步展开讲解这些概念，并在其基础之上构建一个更复杂的聊天应用。

我们最终要构建出的应用如图13-1所示。

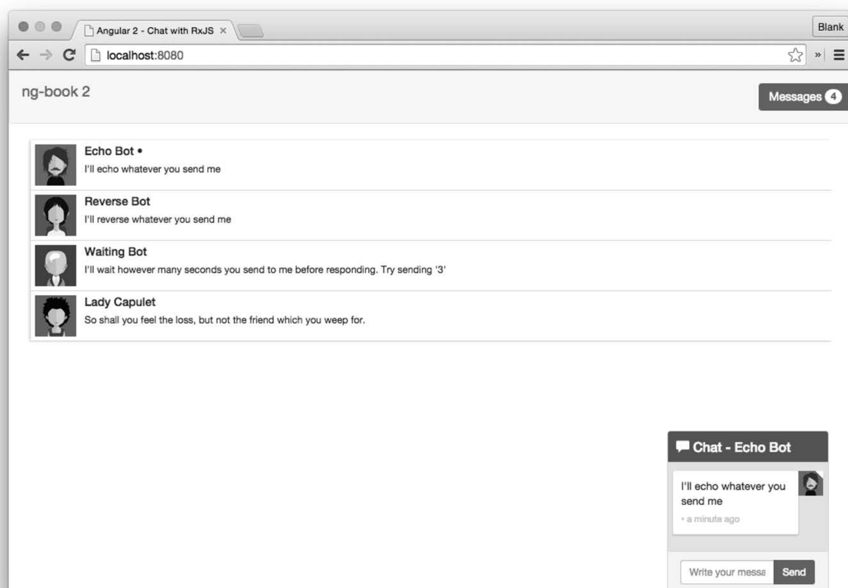


图13-1　完成后的聊天应用

13.1　阅读背景

在第10章和第11章中，我们用RxJS构建了一个聊天应用。我们打算再构建一个完全相同的应

用，但这次改用Redux。这样你就能对比同一个应用在不同数据架构策略下的实现方式了。

你不用为阅读本章的内容而先去阅读第10章和第11章，它们是相互独立的。如果你已经读过了那两章，就可以跳过本章中代码相同的那部分内容（比如，数据模型本身并没有什么变化）。

不过我们确实希望你先读完第12章或至少比较熟悉Redux。

13.2　聊天应用概览

这个应用提供了几个机器人，你可以和它们聊天。先运行这些代码看看：

```
cd code/redux/angular2-redux-chat
npm install
npm run go
```

现在在浏览器中打开http://localhost:8080。



如果上面的链接无法打开，请尝试这个链接：http://localhost:8080/webpack-dev-server/index.html。



一些Windows用户在这个目录下运行npm install时可能会遇到问题。如果遇到了，请先确保自己是在Cygwin[①]中运行这些命令行。

在本应用中，你要注意以下几点：

- 你可以点击会话（thread）和另一个机器人聊天；
- 机器人会根据各自的性格来回复你的消息；
- 右上角的未读消息总数会自动同步。

下面来看看本应用是如何构造的。我们有：

- 三个顶层Angular组件
- 三个数据模型
- 两个reducer及其各自的action creator

我们来逐个看看。

13.2.1　组件

将页面分解成三个顶层组件，如图13-2所示。

① https://www.cygwin.com/

- ChatNavBar：包含未读消息数。
- ChatThreads：展示一个可点击的会话列表，每个会话都包含最新消息和会话头像。
- ChatWindow：展示当前会话的消息和一个用来发送新消息的输入框。

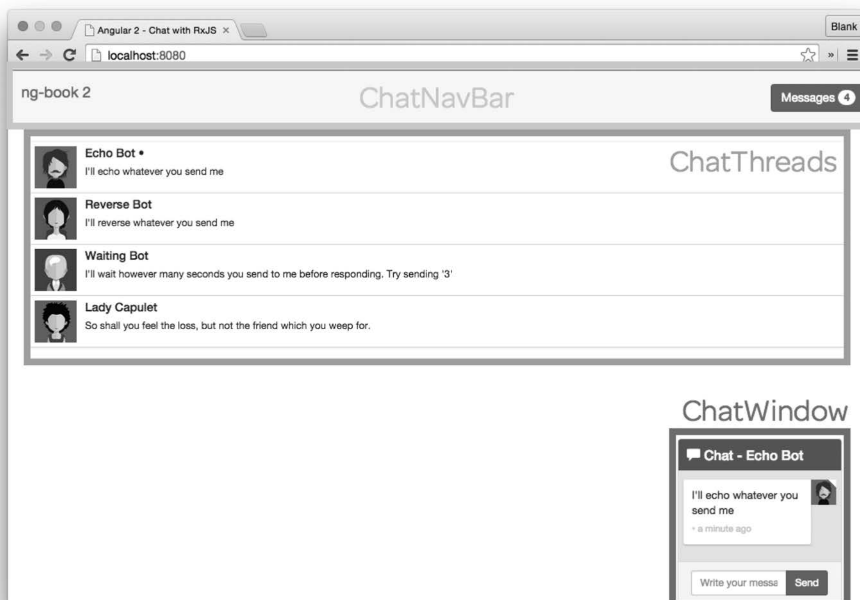


图13-2　Redux聊天应用的顶层组件

13.2.2　数据模型

本应用同样包含三个数据模型，如图13-3所示。

- User：存储聊天参与者的相关信息。
- Message：存储一条单独的信息。
- Thread：存储一组消息的集合以及一些与这次会话有关的其他数据。

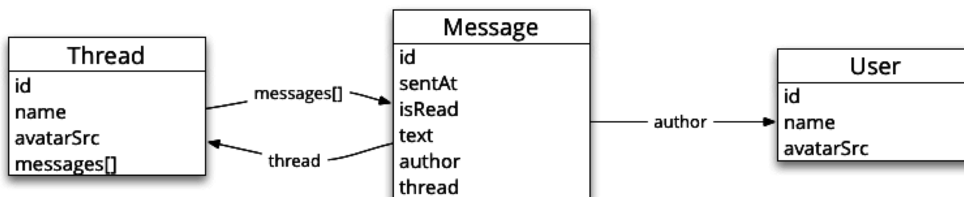


图13-3　Redux聊天应用的数据模型

13.2.3 reducer

本应用有两个reducer。

- ❑ `UsersReducer`：处理当前用户的相关信息。
- ❑ `ThreadsReducer`：处理会话及其相关的消息。

13.2.4 总结

大体来说，本应用的数据架构是这样的：

- ❑ 所有用户和会话（它保存着该会话的消息列表）相关的信息都保存在中心store之中；
- ❑ 组件订阅store的变化并显示合适的数据（未读消息数、会话列表和消息列表本身）；
- ❑ 当用户发送一条消息时，组件就会向store中分发一个action。

本章其余部分将深入讲解如何用Angular和Redux来实现此应用。我们先实现数据模型，然后看看如何创建应用的state和reducer，最后实现组件。

13.3 实现数据模型

我们先从简单的部分开始，看看数据模型。

我们会用`interface`（接口）来规定每个数据模型的定义。这不是必需的，你也可以使用更复杂一些的对象。尽管如此，带方法的对象可能会改变自己的内部状态，而这会破坏我们努力建立的函数式模型。

也就是说，应用中state的所有变化都只能由reducer发起；state中的对象本身应该是不可变的。

因此，通过把数据模型定义为`interface`，就可以：

(1) 在编译阶段确保我们使用的对象是符合预期格式的；

(2) 减少风险，比如不小心往数据模型对象中添加了某个方法而导致意想不到的行为。

13.3.1 `User`

`User`接口中有`id`、`name`和`avatarSrc`。

code/redux/angular2-redux-chat/app/ts/models/User.ts

```
export interface User {
  id: string;
  name: string;
  avatarSrc: string;
  isClient?: boolean;
}
```

我们还有一个布尔值属性isClient（问号表明这个字段是可选的）。当使用本应用的是人而不是机器人时，我们会把User中的该字段设为true。

13.3.2　Thread

同样，Thread也是一个TypeScript接口。

code/redux/angular2-redux-chat/app/ts/models/Thread.ts

```
export interface Thread {
  id: string;
  name: string;
  avatarSrc: string;
  messages: Message[];
}
```

我们存储了Thread的id、name和avatarSrc，而messages字段中存储的是Message的数组。

13.3.3　Message

Message是第三个也是最后一个数据模型的interface。

code/redux/angular2-redux-chat/app/ts/models/Message.ts

```
export interface Message {
  id?: string;
  sentAt?: Date;
  isRead?: boolean;
  thread?: Thread;
  author: User;
  text: string;
}
```

每条消息都包含以下内容。

- id：消息的id。
- sentAt：消息的发送时间。
- isRead：一个布尔值标识，表示消息是否已读。
- author：写这条消息的User。
- text：消息的文本内容。
- thread：对包含这条消息的Thread的引用。

13.4　应用的 state

现在有了数据模型，我们再来讨论一下中心state的模型。在前一章中，我们的中心state是一个对象。它有一个counter键，值的类型是一个number。然而这个应用的state就要复杂多了。

下面是应用state的第一部分。

code/redux/angular2-redux-chat/app/ts/reducers/index.ts

```
export interface AppState {
  users: UsersState;
  threads: ThreadsState;
}
```

AppState也是一个interface，它有两个顶级的键：users和threads。这两个键本身是通过两个接口UsersState和ThreadsState来定义的，而这两个接口是在它们各自的reducer文件中定义的。

13.4.1 关于代码布局

在Redux应用中，一种常用的模式是：顶级state中的每个reducer都对应一个顶级的键。这个应用的顶级reducer在reducers/index.ts文件中。

每个reducer都有自己的文件。每个文件中都有如下内容：

- 用来描述state树当前分支的interface；
- state树当前分支的初始值；
- reducer本身；
- 任何用来查询state树当前分支的选择器——我们还没有讨论过选择器，但是很快就要讲到了。

我们之所以把所有这些截然不同的东西放在一起，是因为它们都是用来处理state树的当前分支的。通过把这些都放在同一个文件中，可以很容易地同时对它们进行重构。

只要愿意，你完全可以使用多级嵌套的布局。如果要分解应用中的大型模块，这是一种很好的方式。

13.4.2 根 reducer

讨论到如何拆分reducer，我们来看看根reducer。

code/redux/angular2-redux-chat/app/ts/reducers/index.ts

```
export interface AppState {
  users: UsersState;
  threads: ThreadsState;
}

const rootReducer: Reducer<AppState> = combineReducers<AppState>({
  users: UsersReducer,
  threads: ThreadsReducer
});
```

注意这里的对称性：UsersReducer作用于users键，而users键的类型是UsersState；ThreadsReducer作用于threads键，而threads键的类型是ThreadsState。

combineReducers让这一切成为可能。它接收一个由键和reducer组成的映射表（map）并返回一个新的reducer，这个新的reducer可以根据这些键进行相应的操作。

当然，我们还没看完AppState的结构，现在继续。

13.4.3　UserState

UsersState保存了currentUser的一个引用。

code/redux/angular2-redux-chat/app/ts/reducers/UsersReducer.ts

```
export interface UsersState {
  currentUser: User;
};

const initialState: UsersState = {
  currentUser: null
};
```

想象一下，state树的这条分支其实可以存储与用户有关的任何信息，比如最后上线的时间、空闲时间等。不过目前这样就足够了。

下面我们会在定义reducer时使用initialState，但此刻只是把当前用户设置为null。

13.4.4　ThreadsState

来看一下ThreadsState。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export interface ThreadsEntities {
  [id: string]: Thread;
}

export interface ThreadsState {
  ids: string[];
  entities: ThreadsEntities;
  currentThreadId?: string;
};

const initialState: ThreadsState = {
  ids: [],
  currentThreadId: null,
  entities: {}
};
```

首先定义了接口ThreadsEntities。它是一个键为会话id，值为会话的映射表。这样我们就

能在这个映射表中通过id找到任意一个会话了。

在`ThreadsState`中还存储了一个名叫`ids`的数组。它用来存储在`entities`中能找到的所有会话的id列表。



常用类库normalizr[1]用到了这种策略。它的理念是，一旦标准化了在Redux的state中存储实体的方式，就可以建造辅助类库并清晰地使用它了。使用了normalizr之后，我们就有了大量的选择来让工作更高效，而不必了解每个state树的格式。

我决定不在本章中讲解normalizr，因为还有许多其他东西要学。不过我确实很喜欢在产品级应用中使用normalizr。

另外，normalizr是完全可选的，即使不在本应用中使用也不会导致任何重大的变化。

如果要学习normalizr，请查阅官方文档https://github.com/paularmstrong/normalizr、博客https://medium.com/@mcowpercoles/using-normalizr-js-in-a-redux-store-96ab33991369#.l8ur7ipu6和Redux作者Dan Abramov在Twitter上的转发https://twitter.com/dan_abramov/status/663032263702106112。

我们用`currentThreadId`保存正在浏览的会话id，目的是了解用户正在浏览的是哪个会话。

把`initialState`都设置为“空值”。

13.4.5　可视化 AppState

Redux DevTools为我们提供了一个Chart视图，它可以让我们检查应用的state。图13-4展示了启动后的所有演示数据。

① https://github.com/paularmstrong/normalizr

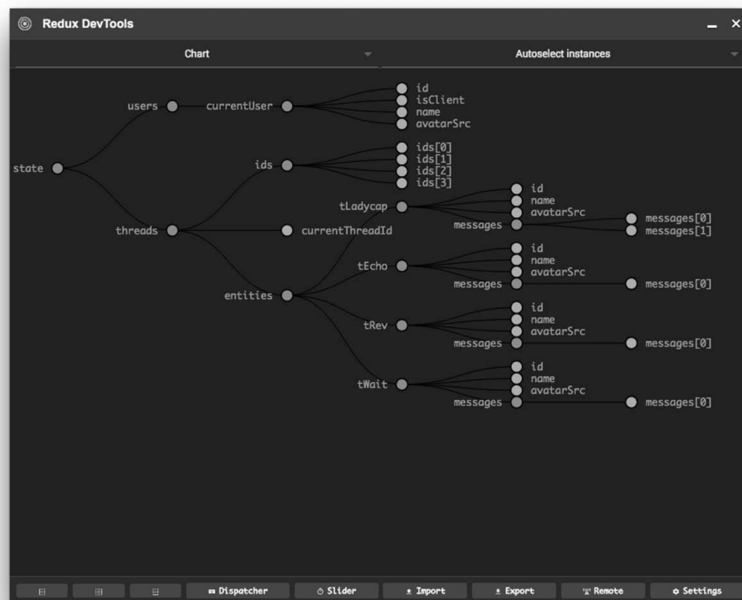


图13-4　Redux聊天应用的状态图

更棒的是可以把鼠标悬停在单个节点上来查看这条数据的各个属性，如图13-5所示。

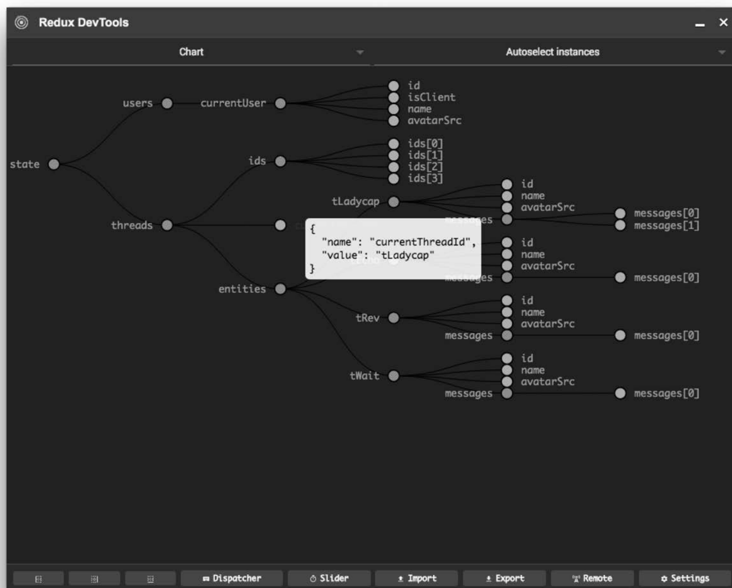


图13-5　查看当前回话

13.5　构建 reducer（和 action creator）

有了中心state，就可以用reducer来改变它了！

既然reducer要处理action，我们就要知道reducer中action的格式。因此在构建reducer的同时也把action creator构建出来。

13.5.1　设置当前用户的 action creator

`UserState`中存储着当前用户，因此需要一个action来设置当前用户。我们会在actions文件夹中保存这些action文件，并且文件名要和它们对应的reducer保持一致，比如在这个例子中的文件名是UserActions。

code/redux/angular2-redux-chat/app/ts/actions/UserActions.ts

```
export const SET_CURRENT_USER = '[User] Set Current';
export interface SetCurrentUserAction extends Action {
  user: User;
}
export const setCurrentUser: ActionCreator<SetCurrentUserAction> =
  (user) => ({
    type: SET_CURRENT_USER,
    user: user
  });
```

这里定义了`const SET_CURRENT_USER`。我们将在reducer的`switch`语句中使用它。

我们还定义了一个新的子接口`SetCurrentUserAction`，它继承了`Action`并添加了一个`user`属性。我们会用`user`属性表明要把哪个用户作为当前用户。

函数`setCurrentUser`就是我们的action creator函数。它接收一个`user`参数并返回一个`SetCurrentUserAction`。我们要把这个返回值传给reducer的action。

13.5.2　`UsersReducer`：设置当前用户

现在我们把视线转向`UsersReducer`。

code/redux/angular2-redux-chat/app/ts/reducers/UsersReducer.ts

```
export const UsersReducer =
  function(state: UsersState = initialState, action: Action): UsersState {
  switch (action.type) {
    case UserActions.SET_CURRENT_USER:
    const user: User = (<UserActions.SetCurrentUserAction>action).user;
      return {
        currentUser: user
      };
    default:
```

```
      return state;
    }
  };
```

和所有reducer一样，UsersReducer返回一个新的state。在这个例子中，它的类型是UsersState。

接下来对action.type使用switch语句，然后处理UserActions.SET_CURRENT_USER分支。

为了设置当前用户，我们需要从输入的action中获取user。为了做到这一点，首先要把action转换成UserActions.SetCurrentUserAction，然后读取它的.user字段。



这似乎有点奇怪。我们本来已经创建了SetCurrentUserAction，但现在switch语句中使用的却是字符串type，并不是直接使用类型。
实际上，这是受TypeScript所迫。当TypeScript被编译成JavaScript后会丢失接口的元数据。我们也可以尝试使用一些反射机制（装饰器元数据或构造函数等等）来实现。
在分发的时候将SetCurrentUserAction转换成Action，在这里又要转换回去，这样确实不够优雅；但对于这个应用来说，这是处理“多态分发”的一种简便做法。

我们需要返回一个新的UserState。因为UserState只有一个键，所以相应的结果对象只有currentUser键并用所传入action中的user属性作为值。

13.5.3　会话和消息概览

这个应用的核心是会话中的消息。我们需要实现三个action：

(1) 往state中添加一个新会话；

(2) 往会话中添加消息；

(3) 选择一个会话。

我们先来创建一个新会话。

13.5.4　添加新会话的 action creator

下面是用来往state中添加新会话的action creator。

code/redux/angular2-redux-chat/app/ts/actions/ThreadActions.ts

```
export const ADD_THREAD = '[Thread] Add';
export interface AddThreadAction extends Action {
  thread: Thread;
}
export const addThread: ActionCreator<AddThreadAction> =
```

```
  (thread) => ({
    type: ADD_THREAD,
    thread: thread
  });
```

注意，它在结构上和我们的前一个action creator非常相似。我们定义了一个用在switch语句中的常量ADD_THREAD、一个自定义的Action和一个用来生成Action的action creator addThread。

注意，这里并没有初始化Thread本身，因为这个Thread是作为参数传进来的。

13.5.5 添加新会话的 reducer

现在通过处理ADD_THREAD分支来创建ThreadsReducer。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const ThreadsReducer =
  function(state: ThreadsState = initialState, action: Action): ThreadsState {
  switch (action.type) {

    // Adds a new Thread to the list of entities
    case ThreadActions.ADD_THREAD: {
      const thread = (<ThreadActions.AddThreadAction>action).thread;

      if (state.ids.includes(thread.id)) {
        return state;
      }

      return {
        ids: [ ...state.ids, thread.id ],
        currentThreadId: state.currentThreadId,
        entities: Object.assign({}, state.entities, {
          [thread.id]: thread
        })
      };
    }

    // Adds a new Message to a particular Thread
```

ThreadsReducer处理的是ThreadsState。当处理ADD_THREAD这个action时，我们把action对象类型又转换回了ThreadActions.AddThreadAction并从中取出thread。

接着检查在state.ids的列表中是否包含这个新的thread.id。如果已经有了，那么就不作任何改动，直接返回当前的state。

但如果这个thread是新的，那就要把它添加到当前的state中。

记住，创建一个新的ThreadsState时要格外小心，不要修改旧的state。这个state比我们以前接触过的要复杂得多，但在处理原则上是基本一致的。

我们先把thread.id添加到ids数组中。这里使用了ES6的展开操作符（...）来表明我们想

把所有现存的`state.ids`放入新数组之中并在数组结尾处添加`thred.id`。

添加一个新会话时`currentThreadId`并没有改变，所以这里直接返回原来的`state.currentThreadId`即可。

对于`entities`，需要记住的是它是一个对象。它的键是每个会话的id字符串，值是这个会话本身。这里使用`Object.assign`来创建一个新对象，新对象中包含了老的`state.entities`和一个新的`thread`对象。



每次进行修改时都要小心翼翼地复制这些对象是不是让你觉得很烦？别人也都这么想！事实上，这样做会很容易意外修改原始数据。
这也就是出现Immutable.js[1]的原因了。和Redux一起使用Immutable.js通常就是出于这个目的。Immutable会帮我们处理好这些原本需要小心进行的更新。
我建议你查看Immutable.js，看看它对写reducer是否更合适。

现在就可以把新会话添加到中心state里了！

13.5.6　添加新消息的 action creator

有了会话，我们就可以开始往里面添加消息了。

为添加消息定义一个新的action。

code/redux/angular2-redux-chat/app/ts/actions/ThreadActions.ts

```
export const ADD_MESSAGE = '[Thread] Add Message';
export interface AddMessageAction extends Action {
  thread: Thread;
  message: Message;
}
```

`AddMessageAction`往会话中添加一条消息。

下面是添加新消息的action creator。

code/redux/angular2-redux-chat/app/ts/actions/ThreadActions.ts

```
export const addMessage: ActionCreator<AddMessageAction> =
  (thread: Thread, messageArgs: Message): AddMessageAction => {
    const defaults = {
      id: uuid(),
      sentAt: new Date(),
      isRead: false,
      thread: thread
    };
```

① https://facebook.github.io/immutable-js/

```
    const message: Message = Object.assign({}, defaults, messageArgs);

    return {
      type: ADD_MESSAGE,
      thread: thread,
      message: message
    };
  };
```

addMessage这个action creator接收一个thread和一个准备加工成消息的对象。注意，这里保留了一个defaults的列表，目的是把创建id、设置时间戳和设置isRead状态等操作封装起来。对于发送信息的人来说，这样就完全不用关心UUID的具体格式是什么了。

如果用户已经事先用UUID类库创建好了自带id的消息，当用户发送这条消息时，我们也会将它保存起来。为了实现这种默认行为，先把messageArgs合并到defaults之中，再合并到一个新的对象中。

最后，我们返回了带有thread和新的message且类型为ADD_MESSAGE的action。

13.5.7　添加新消息的 reducer

现在我们要在ThreadsReducer中添加ADD_MESSAGE的处理器。要添加一条新消息，我们就要获得这个会话，然后把消息添加到这个会话中。

这里还有微妙的一点要处理：如果该thread是当前会话，那就要将这条消息标记为已读。

用户永远都会有一个会话是当前会话，也就是他们正在查看的会话。我们的意思是，如果把一条新消息添加到了当前会话中，那么它就会被自动标记为已读。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
  case ThreadActions.ADD_MESSAGE: {
    const thread = (<ThreadActions.AddMessageAction>action).thread;
    const message = (<ThreadActions.AddMessageAction>action).message;

    // special case: if the message being added is in the current thread, then
    // mark it as read
    const isRead = message.thread.id === state.currentThreadId ?
                    true : message.isRead;
    const newMessage = Object.assign({}, message, { isRead: isRead });

    // grab the old thraed from entities
    const oldThread = state.entities[thread.id];

    // create a new thread which has our newMessage
    const newThread = Object.assign({}, oldThread, {
      messages: [...oldThread.messages, newMessage]
    });

    return {
```

```
        ids: state.ids, // unchanged
        currentThreadId: state.currentThreadId, // unchanged
        entities: Object.assign({}, state.entities, {
          [thread.id]: newThread
        })
      };
    }

    // Select a particular thread in the UI
```

这段代码有点长，因为我们要小心地避免修改原来的会话，但它大体上和我们以前所做的没什么不同。

首先，提取出`thread`和`message`。

如果这条消息属于当前会话（接下来就会看到如何设置当前会话），我们就把它标记为已读。

然后，我们抓取`oldThread`并把`newMessage`追加到旧的`messages`数组，以创建`newThread`。

最后，我们返回新的`ThreadsState`。当前的会话`ids`列表和`currentThreadId`在添加一条新消息时都没有变，所以这里直接使用原有值。唯一改变的就是我们用`newThread`更新了`entities`。

现在来实现我们数据骨架的最后一部分：选择会话。

13.5.8　选择会话的 action creator

用户可以同时进行多个聊天会话，但是只有一个聊天窗口（也就是用户可以阅读和发送消息的地方）。当用户点击了一个会话，我们就要在聊天窗口中展示这个会话中的消息，如图13-6所示。

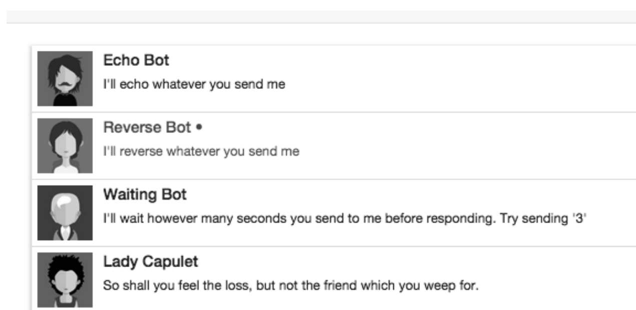


图13-6　选择一个会话

我们需要记录哪个会话是当前选中的会话。要做到这一点，需要使用`ThreadsState`中的`currentThreadId`属性。

我们来创建它的action。

code/redux/angular2-redux-chat/app/ts/actions/ThreadActions.ts

```
export const SELECT_THREAD = '[Thread] Select';
export interface SelectThreadAction extends Action {
  thread: Thread;
}
export const selectThread: ActionCreator<SelectThreadAction> =
  (thread) => ({
    type: SELECT_THREAD,
    thread: thread
  });
```

这个action中并没有引入新概念，只有新的动作类型`SELECT_THREAD`和当前选中并作为参数传入的`thread`。

13.5.9　选择会话的 reducer

选择一个`thread`需要做两件事：

(1) 把`currentThreadId`设置为选中`thread`的`id`；

(2) 把这个`thread`中的所有消息标记为已读。

下面是这个reducer的代码。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
    case ThreadActions.SELECT_THREAD: {
      const thread = (<ThreadActions.SelectThreadAction>action).thread;
      const oldThread = state.entities[thread.id];

      // mark the messages as read
      const newMessages = oldThread.messages.map(
        (message) => Object.assign({}, message, { isRead: true }));

      // give them to this new thread
      const newThread = Object.assign({}, oldThread, {
        messages: newMessages
      });

      return {
        ids: state.ids,
        currentThreadId: thread.id,
        entities: Object.assign({}, state.entities, {
          [thread.id]: newThread
        })
      };
    }

    default:
```

```
      return state;
  }
};
```

首先获取要选择的thread然后使用thread.id从state中得到当前会话的值。



这是个防御型策略。为什么不直接使用传进来的thread呢？对于一些应用来说这也许是正确的设计决策。但在这个例子中，需要通过读取state.entities中会话的最后一个已知值来使thread免受外部修改。

接下来，我们创建所有旧消息的副本并把它们全部设置为isRead: true。然后把新的已读消息列表赋给newThread。

最后，我们返回新的ThreadsState。

13.5.10 reducer 总结

完成了！这些就是搭建数据架构的骨架所需的一切。

回顾一下，UsersReducer负责维护当前用户，而ThreadsReducer则负责管理：

- 会话列表
- 会话中的消息列表
- 当前选中的会话

我们可以从这些数据中拿到所需的一切了（比如未读消息数）。

接下来就把它们和组件连接在一起！

13.6 构建 Angular 聊天应用

如前所述，页面会被分解成三个顶层组件，如图13-7所示。

- ChatNavBar：包含未读消息数。
- ChatThreads：展示一个可点击的会话列表，每个会话包含最后一条消息和会话头像。
- ChatWindow：展示当前会话的消息和一个用来发送新消息的输入框。

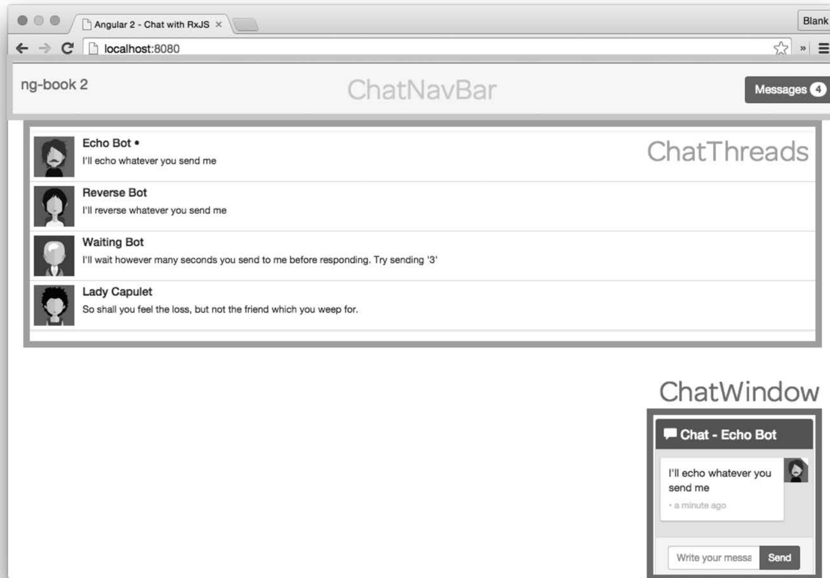


图13-7 Redux聊天应用的顶层组件

我们要像上一章一样启动本应用。在应用的最上层，我们初始化Redux store并通过Angular的依赖注入系统来提供它。（如果觉得陌生，请重新阅读上一章。）

code/redux/angular2-redux-chat/app/ts/app.ts

```
let store: Store<AppState> = createStore<AppState>(
  reducer,
  compose(devtools)
);

@NgModule({
  declarations: [
    ChatApp,
    ChatPage,
    ChatThreads,
    ChatNavBar,
    ChatWindow,
    ChatThread,
    ChatMessage,
    FromNowPipe
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  bootstrap: [ ChatApp ],
  providers: [
```

```
    { provide: AppStore, useFactory: () => store }
  ]
})
class ChatAppModule {}

platformBrowserDynamic().bootstrapModule(ChatAppModule)
```

13.6.1　顶层组件 ChatApp

ChatApp是顶层组件，只负责渲染ChatPage组件。

code/redux/angular2-redux-chat/app/ts/app.ts

```
@Component({
  selector: 'chat-app',
  template: `
  <div>
    <chat-page></chat-page>
  </div>
  `
})
class ChatApp {
  constructor(@Inject(AppStore) private store: Store<AppState>) {
    ChatExampleData(store);
  }
}
```



这个应用中机器人的数据来自客户端而不是服务器端。ChatExampleData()函数为应用设置了初始数据。我们不会在本书中具体解释这段代码，如果你想了解它的工作细节，可以随时查阅源代码。

我们没有在这个应用中使用路由。如果要用的话，可以把与路由相关的内容放到应用的顶层组件之中。现在只创建ChatPage组件来渲染应用的主体部分。

这个应用中没有其他页面，但为每个页面分配一个组件仍然是个好主意，毕竟将来万一还要添加其他页面呢。

13.6.2　ChatPage

聊天页面会渲染三个主要组件：

- ChatNavBar
- ChatThreads
- ChatWindow

下面是其代码。

code/redux/angular2-redux-chat/app/ts/pages/ChatPage.ts

```
@Component({
  selector: 'chat-page',
  template: `
  <div>
    <chat-nav-bar></chat-nav-bar>
    <div class="container">
      <chat-threads></chat-threads>
      <chat-window></chat-window>
    </div>
  </div>
  `
})
export default class ChatPage {
}
```

我们在这个应用中使用的是一种叫作容器型组件的设计模式。这三个组件都是容器型组件。下面就来解释一下。

13.6.3 容器型组件与展示型组件

如果数据散布于所有组件中，那么这个应用就会很难理解。然而，我们的应用是动态的，组件需要运行时的数据来填充，也需要响应用户的交互。

缓解这种冲突的模式之一就是区分展示型组件与容器型组件的概念。具体来说是这样的：

(1) 要让与外部数据源（例如API、Redux store、Cookies等）交互的组件尽可能少；

(2) 要有意识地将数据访问放在容器型组件之中；

(3) 对于纯“功能性”的展示型组件，要求它的所有属性（输入和输出）都由容器型组件来管理。

这种设计的好处在于展示型组件的行为是可预测的。它们可以复用，因为它们只关心自己用到的那部分数据，从不对整体的数据架构作出任何假设。

即使不考虑可复用性，其可预测性也是一个优点。对于相同的输入，它们总是会给出相同的输出（比如用相同的方式渲染）。



仔细想想，你会发现要求reducer必须是纯函数和要求展示型组件必须是“纯组件”背后的哲学是一样的。

如果整个应用全都是展示型组件，那是最理想的。但现实世界中的数据是杂乱、不断变化的，所以我们可以试着把用来适应真实世界的各种复杂数据封装到容器型组件中。



如果你是高级程序员，可能会发现在MVC和容器/展示型组件之间存在着一种不太准确的比喻。也就是说，展示型组件类似于所传入数据的“视图”，而容器型组件则类似于“控制器”，它接收“数据模型”（应用其他部分的数据）并在进行适配之后传给展示型组件。

但如果你还是编程界的新兵，那就先别试图理解“Angular组件本身就是视图和控制器”这种比喻了。

在这个应用中，容器型组件就是那些和store交互的组件。这表示容器型组件符合下列三种特征：

(1) 从store中读取数据；

(2) 订阅store的变化；

(3) 向store中分发action。

这里的三个主要组件都是容器型组件，而它们所包含的组件都是展示型的（也就是功能性的/纯粹的/不和store交互的）。

接下来构建第一个容器型组件：导航条。

13.7　构建 ChatNavBar

导航条中要显示当前用户的未读消息数，如图13-8所示。

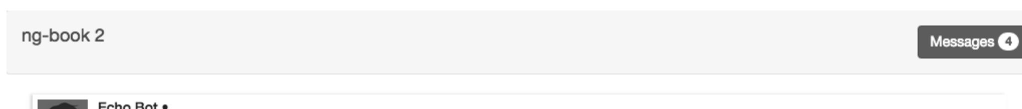


图13-8　ChatNavBar组件中的未读数



试验未读消息数量最好的办法是使用等待机器人（Waiting Bot）。如何你还没有试过，尝试发消息“3”给等待机器人，然后切换到其他聊天窗口。等待机器人会等3秒再给你回复消息，这样你就会看到未读消息数量的增长。

先来看看组件代码。

code/redux/angular2-redux-chat/app/ts/containers/ChatNavBar.ts

```
@Component({
  selector: 'chat-nav-bar',
  template: `
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
```

```
        <a class="navbar-brand" href="https://ng-book.com/2">
          <img src="${require('images/logos/ng-book-2-minibook.png')}"/>
           ng-book 2
        </a>
      </div>
      <p class="navbar-text navbar-right">
        <button class="btn btn-primary" type="button">
          Messages <span class="badge">{{ unreadMessagesCount }}</span>
        </button>
      </p>
    </div>
  </nav>
  `
})
export default class ChatNavBar {
  unreadMessagesCount: number;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.updateState());
    this.updateState();
  }

  updateState() {
    this.unreadMessagesCount = getUnreadMessagesCount(this.store.getState());
  }
}
```

模板为我们提供了DOM结构和渲染导航条所需的CSS（这些CSS类来自CSS框架Bootstrap）。

在这个模板中，我们唯一要显示的变量是unreadMessagesCount。

ChatNavBar组件中的unreadMessagesCount是一个实例变量。它会被设置成所有会话的未读消息总数。

注意，我们在constructor中做了三件事：

(1) 注入了store；

(2) 订阅了store中的任何变化；

(3) 调用了this.updateState()。

我们在subscribe后调用了this.updateState()，因为要确保组件使用最新数据进行初始化。subscribe只会在组件初始化之后state数据发生变化的时候调用。

updateState()是最有意思的函数——我们把unreadMessagesCount设置为getUnreadMessagesCount函数的返回值。getUnreadMessagesCount是什么？它从哪里来？

getUnreadMessagesCount是一个名叫选择器（selector）的新概念。

13.7.1　Redux 选择器

思考一下AppState，我们该如何获取未读消息总数呢？像下面这样如何：

```
// get the state
let state = this.store.getState();

// get the threads state
let threadsState = state.threads;

// get the entities from the threads
let threadsEntities = threadsState.entities;

// get all of the threads from state
let allThreads = Object.keys(threadsEntities)
              .map((threadId) => entities[threadId]);

// iterate over all threads and ...
let unreadCount = allThreads.reduce(
      (unreadCount: number, thread: Thread) => {
        // foreach message in that thread
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            // if it's unread, increment unread count
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0);
```

我们应该把这段逻辑放在ChatNavBar组件中吗？如果这么做的话，会有如下两个问题。

(1) 这一大块代码深深地渗透到了AppState中。更好的方法是把这段逻辑移到所涉及的state之后。

(2) 如果应用的其他地方需要显示未读消息总数呢？如何共享这段逻辑？

选择器背后的思想可用来解决这些问题：

> 选择器是函数，它接收部分state并返回一个值。

我们来看看如何创建选择器。

13.7.2　会话选择器

先从简单的部分开始。假设我们要在AppState中获取ThreadsState。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const getThreadsState = (state): ThreadsState => state.threads;
```

相当简单，对不对？只要给定了顶层的AppState，就可以通过state.threads找到Threads-State。

如果我们要获取当前会话，可以这样做：

```
const getCurrentThread = (state: AppState): Thread => {
  let currentThreadId = state.threads.currentThreadId;
  return state.threads.entities[currentThreadId];
}
```

对于这个小例子来说，这样的选择器就可以胜任。值得考虑的是，如何随着应用的增长让选择器更具可维护性。如果能用选择器来查询其他选择器就好了。如果一个选择器能指定多个其他选择器作为自己的依赖就更好了。

这些正是reselect[1]类库提供的。利用reselect，我们可以创建更小、更专注的选择器，还能结合它们实现更大的功能。

下面来看看如何使用reselect提供的createSelector方法获取当前会话。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const getThreadsEntities = createSelector(
  getThreadsState,
  ( state: ThreadsState ) => state.entities );
```

先来写一个getThreadsEntities选择器。getThreadsEntities使用createSelector并传入两个参数：

(1) 之前定义的选择器getThreadsState；

(2) 一个回调函数，用于接收getThreadsState选择器的返回值，并返回我们要选取的值。

这里只获取了state.entities，看起来似乎有点浪费，但它为我们建立了可维护性更强的选择器。现在看看如何用createSelector创建getCurrentThread。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const getCurrentThread = createSelector(
  getThreadsEntities,
  getThreadsState,
  ( entities: ThreadsEntities, state: ThreadsState ) =>
    entities[state.currentThreadId] );
```

注意，这里引用了两个选择器作为依赖：getThreadsEntities和getThreadsState。这些选择器被解析后就会变成回调函数的参数。我们可以把它们组合起来返回当前选中的会话。

① https://github.com/reactjs/reselect#createselectorinputselectors--inputselectors-resultfunc

13.7.3　未读消息总数选择器

现在我们已经理解了选择器的工作原理，接着就来创建一个选择器以获取未读消息的数量。如果看过前面获取未读消息总数的首次尝试，你会发现每个变量都可以被替换成它们自己的选择器（getThreadsState、getThreadsEntities等）。

下面是用来获取所有Thread的选择器。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const getAllThreads = createSelector(
  getThreadsEntities,
  ( entities: ThreadsEntities ) => Object.keys(entities)
                        .map((threadId) => entities[threadId]));
```

拿到所有会话之后，我们就可以知道所有会话中的未读消息总数。

code/redux/angular2-redux-chat/app/ts/reducers/ThreadsReducer.ts

```
export const getUnreadMessagesCount = createSelector(
  getAllThreads,
  ( threads: Thread[] ) => threads.reduce(
      (unreadCount: number, thread: Thread) => {
        thread.messages.forEach((message: Message) => {
          if (!message.isRead) {
            ++unreadCount;
          }
        });
        return unreadCount;
      },
      0));
```

有了这个选择器，我们就可以在ChatNavBar组件中（以及应用中任何需要的地方）获取到未读消息的数量。

13.8　构建 ChatThreads 组件

接下来在ChatThreads组件中构建会话列表，如图13-9所示。



图13-9　按时间排序的会话列表

13.8.1 ChatThreads 控制器

在查看ChatThreads组件的模板之前，我们先来看看组件的控制器。

code/redux/angular2-redux-chat/app/ts/containers/ChatThreads.ts

```
export default class ChatThreads {
  threads: Thread[];
  currentThreadId: string;

  constructor(@Inject(AppStore) private store: Store<AppState>) {
    store.subscribe(() => this.updateState());
    this.updateState();
  }

  updateState() {
    let state = this.store.getState();

    // Store the threads list
    this.threads = getAllThreads(state);

    // We want to mark the current thread as selected,
    // so we store the currentThreadId as a value
    this.currentThreadId = getCurrentThread(state).id;
  }

  handleThreadClicked(thread: Thread) {
    this.store.dispatch(ThreadActions.selectThread(thread));
  }
}
```

在这个组件中存储了两个实例变量。

- threads：会话列表。
- currentThreadId：用户正在操作的当前会话。

在constructor中保存了一个Redux store的引用并订阅更新。一旦store发生变化，就调用updateState()。

updateState()会保持实例变量与Redux store同步。注意我们正在用的这两个选择器：

- getAllThreads
- getCurrentThread

这样就可以保持它们各自的实例变量总是最新的。

这里引入了一个新概念：事件处理器handleThreadClicked。handleThreadClicked会分发selectThread这个action。当点击一个会话时，我们就告诉store把这个新会话设为所选会话并且应用的其余部分也应该依次更新。

13

13.8.2　ChatThreads 的 template

我们来看一下ChatThreads组件的template及其配置。

code/redux/angular2-redux-chat/app/ts/containers/ChatThreads.ts

```
 */
@Component({
  selector: 'chat-threads',
  template: `
  <!-- conversations -->
  <div class="row">
    <div class="conversation-wrap">
      <chat-thread
           *ngFor="let thread of threads"
           [thread]="thread"
           [selected]="thread.id === currentThreadId"
           (onThreadSelected)="handleThreadClicked($event)">
      </chat-thread>
    </div>
  </div>
  `
```

我们在模板中使用ngFor来遍历threads。我们还用了一个叫作ChatThread的新组件来渲染单个会话。

ChatThread是一个展示型组件。在ChatThread中，我们既不能使用store，也不能读取数据和分发action。反之，我们要通过inputs（输入参数）来传入该组件所需的一切，并通过outputs（输出参数）来处理任何交互。

接着我们会介绍ChatThread的实现，但先来看看这个模板中的输入和输出。

- 使用单个thread变量作为输入属性[thread]；
- 对于输入属性[selected]，我们传入一个布尔值来表明这个会话（thread.id）是否是当前会话（currentThreadId）；
- 如果会话被点击，就发出输出事件(onThreadSelected)。这时就会调用handleThreadClicked()（它会向store中分发选择会话的事件）。

我们再来研究一下ChatThread组件。

13.9　单个 ChatThread 组件

ChatThread组件用来显示会话列表中一个单独的会话。记住ChatThread是展示型组件，它只会操作直接给它的那些数据。

因为它是一个展示型组件，所以我们将它放在app/ts/components文件夹中。

下面是组件控制器的代码。

code/redux/angular2-redux-chat/app/ts/components/ChatThread.ts

```
export default class ChatThread {
  thread: Thread;
  selected: boolean;
  onThreadSelected: EventEmitter<Thread>;

  constructor() {
    this.onThreadSelected = new EventEmitter<Thread>();
  }

  clicked(event: any): void {
    this.onThreadSelected.emit(this.thread);
    event.preventDefault();
  }
}
```

这里的看点是onThreadSelected这个EventEmitter。如果你还没怎么用过EventEmitter，可以把它当作观察者模式的一种实现。我们把它作为这个组件的“输出通道”——想发送数据时就调用onThreadSelected.emit方法，把想要发送的数据传进去。

在这个例子中，我们想把当前会话作为参数传给EventEmitter。当点击这个元素时，我们就会调用onThreadSelected.emit(this.thread)，它会触发父级组件（ChatThreads）中的回调函数。

ChatThread 的@Component 和 template

下面是@Component注解和template的代码。

code/redux/angular2-redux-chat/app/ts/components/ChatThread.ts

```
@Component({
  inputs: ['thread', 'selected'],
  selector: 'chat-thread',
  outputs: ['onThreadSelected'],
  template: `
  <div class="media conversation">
    <div class="pull-left">
      <img class="media-object avatar"
           src="{{thread.avatarSrc}}">
    </div>
    <div class="media-body">
      <h5 class="media-heading contact-name">{{thread.name}}
        <span *ngIf="selected">&bull;</span>
      </h5>
      <small class="message-preview">
        {{thread.messages[thread.messages.length - 1].text}}
      </small>
    </div>
```

```
    <a (click)="clicked($event)" class="div-link">Select</a>
  </div>
  `
```

这里把thread和selected指定为inputs属性，把onThreadSelected指定为outputs属性。

注意，视图中使用了一些直接的绑定，比如{{thread.avatarSrc}}和{{thread.name}}。在class为message-preview的标签中有如下代码：

```
{{ thread.messages[thread.messages.length - 1].text }}
```

它会获取会话中的最后一条消息并显示消息的文本，目的是在每个会话中显示最新消息的预览。

我们还用了*ngIf，会对选中的会话显示•符号。

最后，我们绑定了(click)事件来调用clicked()处理器。注意，我们在调用clicked时传入了参数$event。这是Angular提供的一个用来描述事件的特殊变量。我们通过在clicked处理器中调用event.preventDefault();来使用它。这样可以确保我们不会跳转到其他页面。

13.10　构建 ChatWindow 组件

ChatWindow是这个应用中最复杂的组件（如图13-10所示）。我们一步一步来完成它。

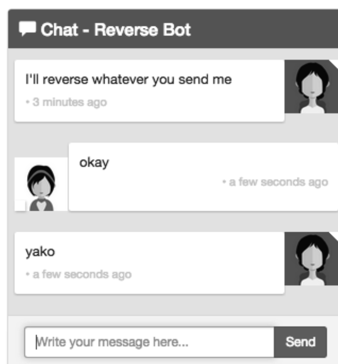


图13-10　聊天窗口

ChatWindow类有三个属性：currentThread（其中包括messages）、draftMessage和currentUser。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
export default class ChatWindow {
  currentThread: Thread;
  draftMessage: { text: string };
  currentUser: User;
```

图13-11表明了每一个属性在何处使用。

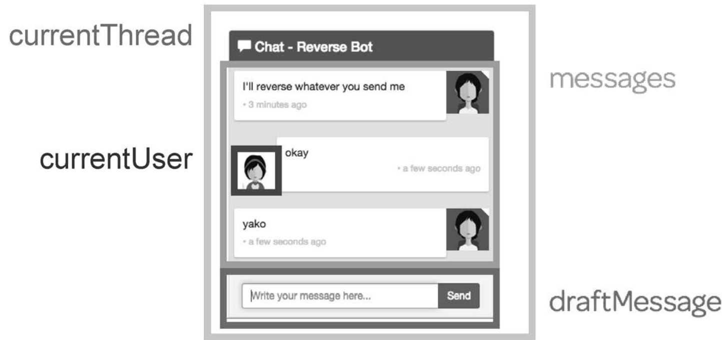


图13-11 聊天窗口的属性

我们在`constructor`中注入了两样东西。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
  constructor(@Inject(AppStore) private store: Store<AppState>,
              private el: ElementRef) {
    store.subscribe(() => this.updateState() );
    this.updateState();
    this.draftMessage = { text: '' };
}
```

第一个是Redux store，第二个是`el`。`el`是一个`ElementRef`，可以用来获取宿主DOM元素。当创建和接收新消息的时候，我们会借助它来让聊天窗口滚动到底部。

我们在构造函数中订阅了store，就像在其他容器型组件中所做的那样。

接着要做的是设置一个默认的`draftMessage`，它的`text`属性是一个空字符串。我们会使用`draftMessage`来记录用户在输入框中输入的消息。

13.10.1 ChatWindow 的 updateState()

当store改变时，我们会更新该组件的实例变量。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
  updateState() {
    let state = this.store.getState();
    this.currentThread = getCurrentThread(state);
    this.currentUser = getCurrentUser(state);
    this.scrollToBottom();
  }
```

我们存储了当前会话和当前用户。如果来了新消息，我们希望滚动到窗口的底部。在这里调

用scrollToBottom有点粗糙，但这种简单的方法可以保证在有新消息时（或用户切换到一个新会话中时）用户不需要每次都手动滚动窗口。

13.10.2　ChatWindow 的 scrollToBottom()

为了滚动到聊天窗口的底部，我们将使用保存在构造函数中的类型为ElementRef的el。要让这个元素滚动，就要设置宿主元素的scrollTop属性。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
scrollToBottom(): void {
  let scrollPane: any = this.el
    .nativeElement.querySelector('.msg-container-base');
  if (scrollPane) {
    setTimeout(() => scrollPane.scrollTop = scrollPane.scrollHeight);
  }
}
```



为什么使用setTimeout?

如果我们得到新消息时立即调用scrollToBottom，那么滚动到底部的动作就是在新消息渲染完成之前执行的。使用setTimeout可以告诉JavaScript我们要在当前执行队列完成后再运行这个函数。该函数会在组件渲染完成**之后**执行，这正是我们想要的效果。

13.10.3　ChatWindow 的 sendMessage

如果我们要发送一条新消息，就要先拿到：

- 当前会话
- 当前用户
- 草稿消息的文本

然后向store中分发一个新的addMessage action。下面是其代码。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
sendMessage(): void {
  this.store.dispatch(ThreadActions.addMessage(
    this.currentThread,
    {
      author: this.currentUser,
      isRead: true,
      text: this.draftMessage.text
    }
  ));
  this.draftMessage = { text: '' };
}
```

sendMessage函数接收draftMessage参数并用组件的属性来设置author和thread。每条已发送的信息其实都已经被读过了（因为是我们写的），所以将其标记为已读。

分发这条消息之后，创建一个新的Message对象并把它赋给this.draftMessage。这会清空输入框。创建一个新对象可以确保我们不会改变已经发送给store的消息。

13.10.4 ChatWindow 的 onEnter

在视图中，我们希望在下面两种场景发送消息：

(1) 用户点击Send按钮；

(2) 用户敲击回车键。

我们定义一个函数来处理这两种事件。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
  onEnter(event: any): void {
    this.sendMessage();
    event.preventDefault();
  }
```



我们创建onEnter事件处理器并把sendMessage作为一个单独的函数，这是因为onEnter要接收一个参数event并调用event.preventDefault()。这种方式下我们还可以在响应浏览器事件之外的场景下调用sendMessage。在这个例子中，我们并没有真的在其他场景下调用sendMessage，但我发现把“真正干活的”函数从事件处理器中独立出来会更好。

否则，sendMessage函数就会：(1) 要求必须传入一个事件对象；(2) 处理该事件对象。但是这样一来它的关注点就太多了。

现在已经处理好了控制器的代码，让我们来看看template。

13.10.5 ChatWindow 的 template

我们先从面板（panel）的起始标签开始，并且在头部显示聊天的名称。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
@Component({
  selector: 'chat-window',
  template: `
    <div class="chat-window-container">
      <div class="chat-window">
        <div class="panel-container">
          <div class="panel panel-default">
```

```
            <div class="panel-heading top-bar">
              <div class="panel-title-container">
                <h3 class="panel-title">
                  <span class="glyphicon glyphicon-comment"></span>
                  Chat - {{currentThread.name}}
                </h3>
              </div>
              <div class="panel-buttons-container" >
                <!-- you could put minimize or close buttons here -->
              </div>
            </div>

            <div class="panel-body msg-container-base">
```

接下来显示消息列表。这里用ngFor遍历消息列表。我们稍后会讲解单个chat-message组件。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
              <chat-message
                   *ngFor="let message of currentThread.messages"
                   [message]="message">
              </chat-message>
            </div>

            <div class="panel-footer">
```

最后是消息输入框和各个结束标签。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
            <div class="input-group">
              <input type="text"
                     class="chat-input"
                     placeholder="Write your message here..."
                     (keydown.enter)="onEnter($event)"
                     [(ngModel)]="draftMessage.text" />
              <span class="input-group-btn">
                <button class="btn-chat"
                   (click)="onEnter($event)"
                   >Send</button>
              </span>
            </div>
          </div>

        </div>
      </div>
    </div>
  </div>
  `
})
export default class ChatWindow {
```

消息输入框是视图中最有意思的部分，我们来看看其中两个有趣的属性：(keydown.enter)和[(ngModel)]。

13.10.6 处理键盘动作

Angular提供了一种简明的方式来处理键盘动作：在元素上绑定事件。在这个例子中，我们绑定了keydown.enter。这表示如果用户按下回车键，就会调用表达式里的函数onEnter($event)。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
                class="chat-input"
                placeholder="Write your message here..."
                (keydown.enter)="onEnter($event)"
                [(ngModel)]="draftMessage.text" />
         <span class="input-group-btn">
```

13.10.7 使用 ngModel

如前所述，Angular并没有像AngularJS那样把双向绑定作为数据架构的核心。特别是当我们使用Redux的时候，它是完全的单向数据流。

然而在组件及其视图之间进行双向绑定是非常有用的。只要把双向绑定的坏处限制在组件之中，保持组件属性和视图的同步是很方便的。

对于这个例子，我们在输入框的值和draftMessage.text之间建立了一个双向绑定。如果在输入框中输入文字，draftMessage.text就会自动设置为输入的文字。同样，如果在代码中更新draftMessage.text，那么视图中输入框的值也会随之改变。

13.10.8 点击 Send 按钮

在Send按钮上将(click)属性绑定到组件中的onEnter函数。

code/redux/angular2-redux-chat/app/ts/containers/ChatWindow.ts

```
                (click)="onEnter($event)"
                >Send</button>
         </span>
```

我们使用同一个onEnter函数来处理本事件。也就是说，点击这个按钮和按回车键都可以发送消息。

13.11 ChatMessage 组件

我们没有把渲染单个消息的代码都放到ChatWindow组件中，而是创建了另一个展示型组件ChatMessage。



提示：如果你发现自己正在使用ngFor，那就表示你该创建一个新组件了。

每条消息都是通过ChatMessage组件渲染的，如图13-12所示。

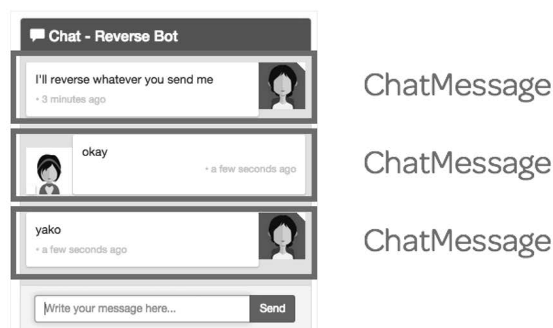


图13-12　ChatMessage组件

该组件相对简明，其主要逻辑是根据消息是否由当前用户所创建来渲染出略有不同的视图。如果该消息不是当前用户创建的，就认为消息是收到的（incoming)。

13.11.1　设置 incoming 属性

记住，每个ChatMessage组件都属于一条Message。因此，要在ngOnInit方法里订阅currentUser流并根据这条Message是否由当前用户创建来设置incoming。

code/redux/angular2-redux-chat/app/ts/components/ChatMessage.ts

```
export default class ChatMessage implements OnInit {
  message: Message;
  incoming: boolean;

  ngOnInit(): void {
    this.incoming = !this.message.author.isClient;
  }
}
```

13.11.2　ChatMessage 的 template

在template中有两点值得注意：

(1) FromNowPipe管道

(2) [ngClass]属性

先来看其代码。

code/redux/angular2-redux-chat/app/ts/components/ChatMessage.ts

```
*/
@Component({
  inputs: ['message'],
  selector: 'chat-message',
  template: `
  <div class="msg-container"
       [ngClass]="{'base-sent': !incoming, 'base-receive': incoming}">

    <div class="avatar"
         *ngIf="!incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>

    <div class="messages"
      [ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}">
      <p>{{message.text}}</p>
      <p class="time">{{message.sender}} • {{message.sentAt | fromNow}}</p>
    </div>

    <div class="avatar"
         *ngIf="incoming">
      <img src="{{message.author.avatarSrc}}">
    </div>
  </div>
  `
```

FromNowPipe是一个管道，把消息的发送时间转换为像“×秒前”这样对用户友好的信息。如你所见，我们要这样使用它：{{message.sentAt | fromNow}}。



FromNowPipe使用优秀的moment.js[①]类库。如果你想学习如何创建自定义管道，可以阅读FromNowPipe的源代码：code/rxjs/chat/app/ts/util/FromNowPipe.ts。

我们也在视图中充分利用了ngClass。当这样写时：

```
[ngClass]="{'msg-sent': !incoming, 'msg-receive': incoming}"
```

我们是在告诉Angular：如果incoming为真就使用msg-receive类（否则使用msg-sent类）。

借助incoming属性，我们就能以不同的形式来显示收到和发出的消息。

13.12 总结

好了，把它们全部放在一起，就是一个完整的聊天应用了（如图13-13所示）!

① http://momentjs.com/

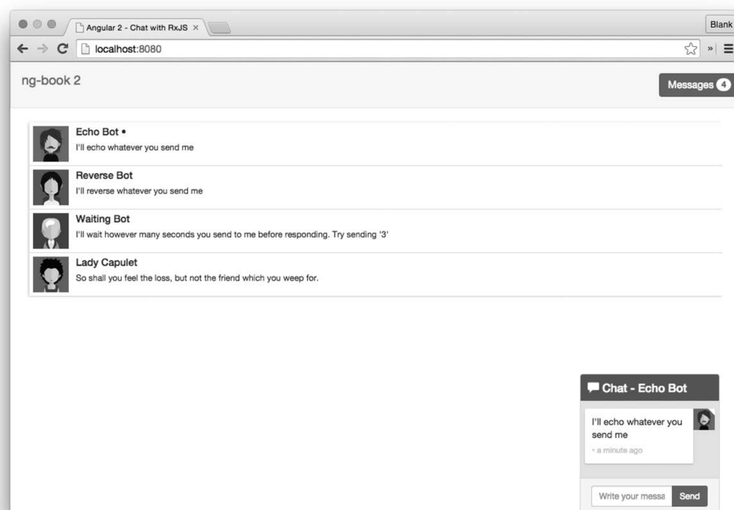


图13-13　完成后的聊天应用

查看文件code/redux/angular2-redux-chat/app/ts/ChatExampleData.ts，你会发现我们已经写好了少量可以跟你聊天的机器人。检出这些代码并试着写几个自己的机器人吧！

第 14 章 高级组件

在本书中，我们已经学习了如何使用Angular的内置指令以及如何创建组件。本章将深入探讨用于开发组件的高级Angular特性。

我们将在本章中学习以下内容：

- 组件样式封装
- 修改宿主DOM元素
- 使用内容投影修改模板
- 访问邻近的指令
- 使用生命周期钩子
- 变更检测

14.1 样式

Angular提供了一套用来指定“组件级”样式的机制。尽管CSS的意思是*层叠样式表*（cascading style sheet），但有时候我们并不想要“层叠”效果。我们可能只想为某个特定的组件提供样式，而不要影响到页面的其他部分。

Angular为组件提供了两个属性来定义CSS类。

为了定义组件样式，我们使用视图属性`styles`来定义内联样式或者借助`styleUrls`属性来使用外部CSS文件，还可以在组件的装饰器中直接定义这些属性。

我们来创建一个使用内联样式的组件。

code/advanced_components/app/ts/styling/styling.ts

```
@Component({
  selector: 'inline-style',
  styles: [`
  .highlight {
    border: 2px solid red;
    background-color: yellow;
```

```
    text-align: center;
    margin-bottom: 20px;
  }
  `],
  template: `
  <h4 class="ui horizontal divider header">
    Inline style example
  </h4>

  <div class="highlight">
    This uses component <code>styles</code>
    property
  </div>
  `
})
class InlineStyle {
}
```

在这个示例中，我们在`styles`数组参数中声明了CSS类`.highlight`，它定义了我们要用的样式。

然后在模板中使用`<div class="highlight">`引用这个类。

最后的结果与我们预期的一样：一个红色边框、黄色背景的`div`（如图14-1所示）。

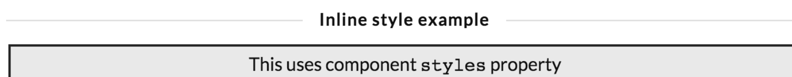


图14-1　使用`styles`属性的组件示例

另一种声明CSS类的方法是使用`styleUrls`属性。它可以让我们从外部文件中定义CSS并在组件中直接引用它们。

在用这种方式创建另一个组件之前，创建一个名为external.css的文件，它包含下面这些类。

code/advanced_components/app/ts/styling/external.css

```
.highlight {
  border: 2px dotted red;
  text-align: center;
  margin-bottom: 20px;
}
```

然后就可以在组件代码中引用它。

code/advanced_components/app/ts/styling/styling.ts

```
@Component({
  selector: 'external-style',
  styleUrls: [externalCSSUrl],
  template: `
  <h4 class="ui horizontal divider header">
    External style example
```

```
  </h4>

  <div class="highlight">
    This uses component <code>styleUrls</code>
    property
  </div>
  `
})
class ExternalStyle {
}
```

加载页面时，就可以看见有虚线边框的div（如图14-2所示）。

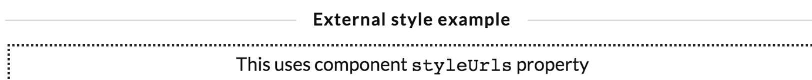


图14-2 使用styleUrls属性的组件示例

14.1.1 视图（样式）封装

这个例子中有意思的地方是，这两个组件都定义了名为highlight的类；尽管其属性是不同的，但它们并没有相互干扰。

这是因为Angular默认将组件样式封装在组件的上下文中。如果检查页面并展开<head>标签，可以注意到Angular把我们定义的样式注入到了一个<style>标签之中，如图14-3所示。

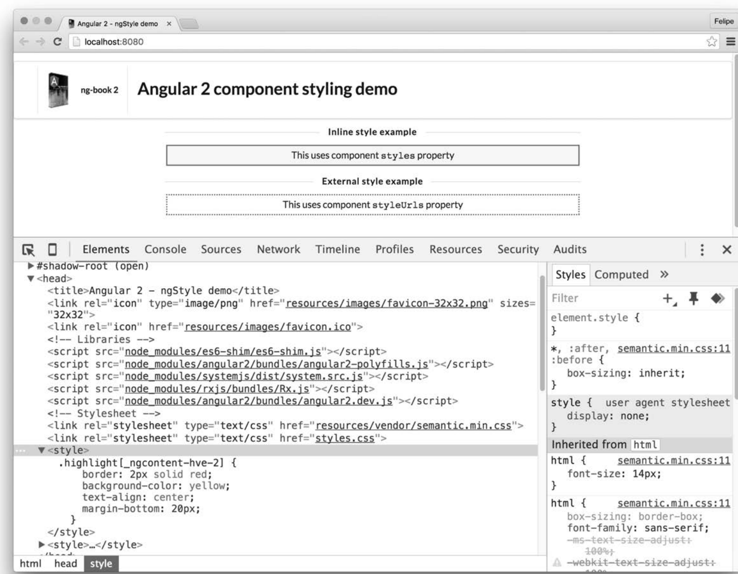


图14-3 注入后的样式

你还会注意到，到这个CSS类使用了_ngcontent-hve-2属性来限定其作用域：

```
.highlight[\_ngcontent-hve-2] {
  border: 2px solid red;
  background-color: yellow;
  text-align: center;
  margin-bottom: 20px; }
}
```

如果查看<div>的渲染结果，会发现它也添加了一个_ng-content-hve-2属性，如图14-4所示。

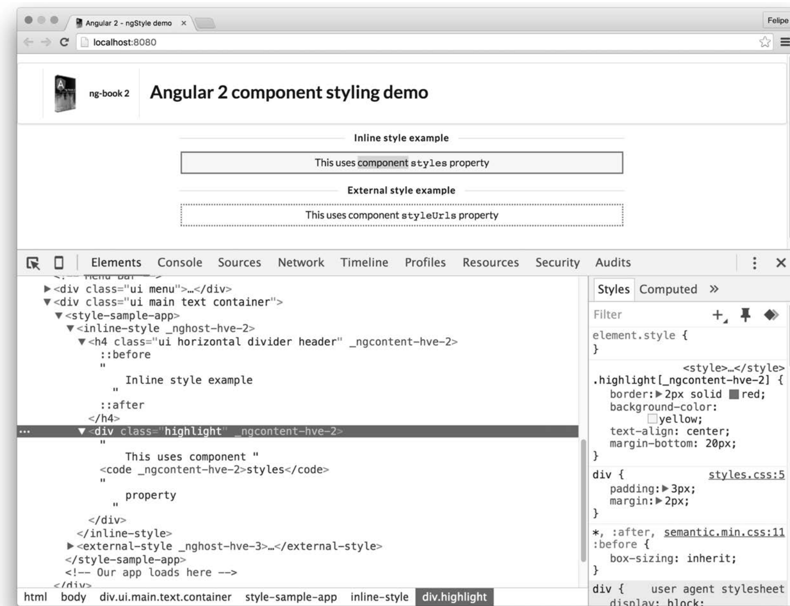


图14-4　注入后的样式：<div>的渲染结果

引用外部样式文件时的效果也是一样的，如图14-5所示。

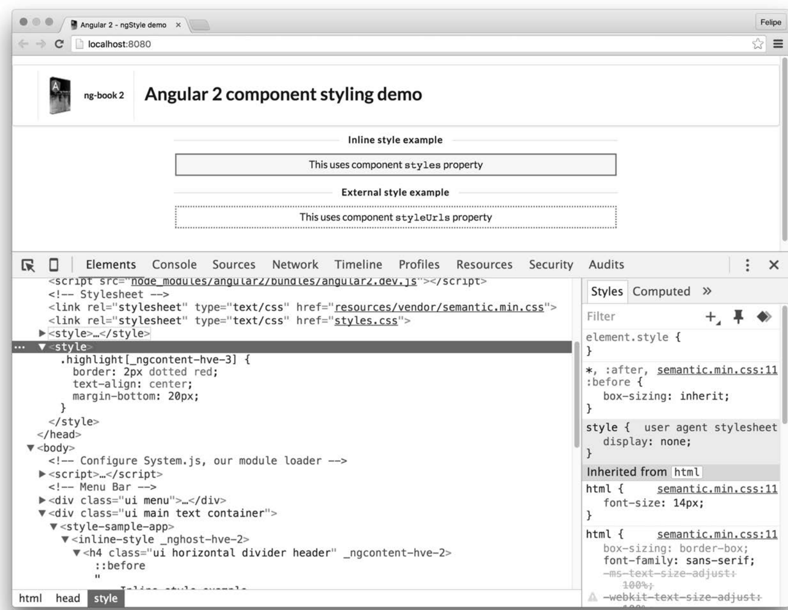


图14-5　外部样式

<div>的渲染结果如图14-6所示。

图14-6　外部样式：<div>的渲染结果

Angular允许我们使用encapsulation属性来更改这种行为。

这个属性可以取下列值之一，它们都定义在ViewEncapsulation枚举中。

- Emulated（仿真）：这是默认选项，它会采用我们刚刚解释过的技术来封装样式。
- Native（原生）：使用这个选项，Angular会采用Shadow DOM技术（下面会详细介绍）。
- None（无）：使用这个选项，Angular不会封装任何样式，允许样式渗透给页面的其他元素。

14.1.2　Shadow DOM 封装

你可能会问：Shadow DOM有什么用呢？通过使用Shadow DOM，组件会生成一棵独一无二的DOM树，而这棵DOM树对于页面中的其他元素是不可见的。这样，在这个元素中定义的样式对页面的其余部分来说就像不存在一样。



要深入了解Shadow DOM，请查阅Eric Bidelman撰写的指南http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/。

我们来创建另一个使用Native封装（Shadow DOM）的组件，理解它是如何工作的。

code/advanced_components/app/ts/styling/styling.ts

```
@Component({
  selector: `native-encapsulation`,
  styles: [`
  .highlight {
    text-align: center;
    border: 2px solid black;
    border-radius: 3px;
    margin-botton: 20px;
  }`],
  template: `
  <h4 class="ui horizontal divider header">
    Native encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.Native</code>
  </div>
  `,
  encapsulation: ViewEncapsulation.Native
})
class NativeEncapsulation {
}
```

在这个例子中，如果查看源代码，会看到如图14-7所示的结果。

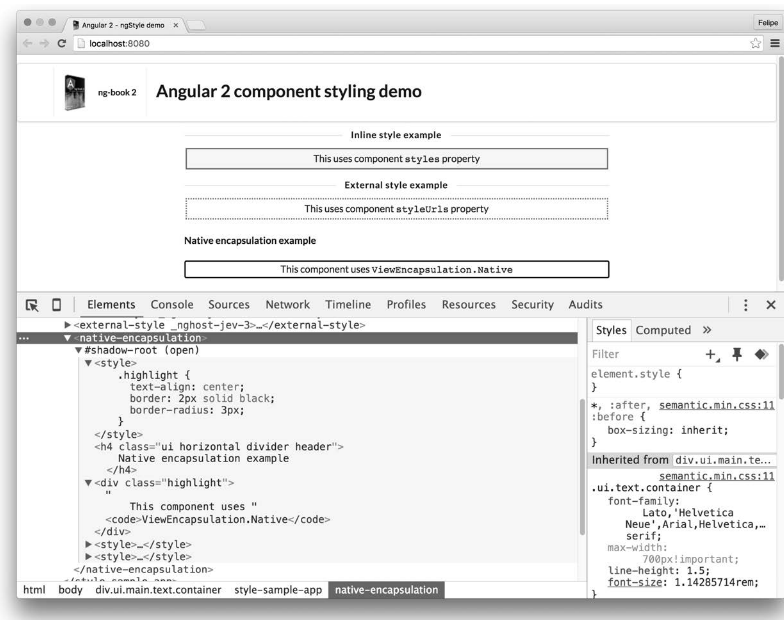


图14-7 Native封装

#shadow-root元素里面的一切都被封装起来了，并且和页面的其他部分是完全隔离的。

14.1.3 不使用封装

最后，如果我们创建一个组件并指定ViewEncapsulation.None，那就不会进行任何的样式封装。

code/advanced_components/app/ts/styling/styling.ts

```
@Component({
  selector: `no-encapsulation`,
  styles: [`
  .highlight {
    border: 2px dashed red;
    text-align: center;
    margin-bottom: 20px;
  }
  `],
  template: `
  <h4 class="ui horizontal divider header">
    No encapsulation example
  </h4>

  <div class="highlight">
    This component uses <code>ViewEncapsulation.None</code>
  </div>
```

```
  `,
  encapsulation: ViewEncapsulation.None
})
class NoEncapsulation {
}
```

检查元素时，会看到如图14-8所示的结果。

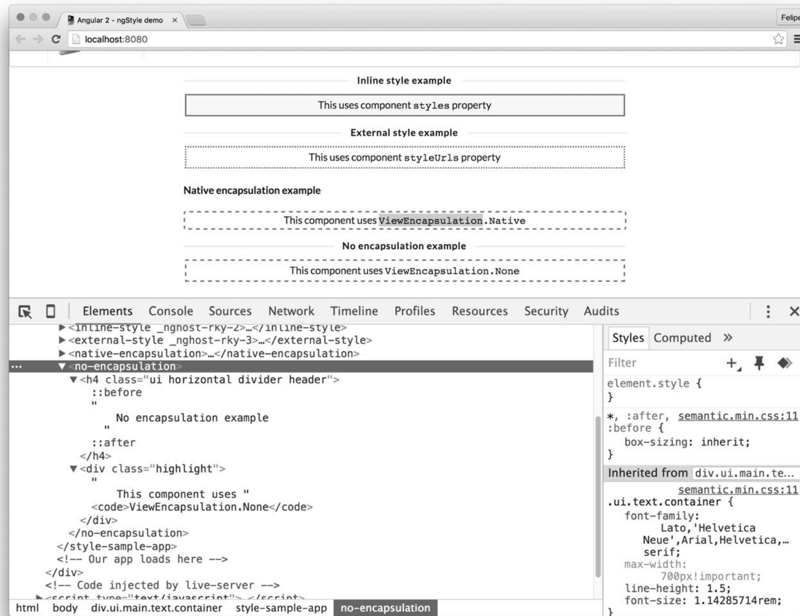


图14-8　不进行封装

可以看到HTML中没有注入任何东西。在页头中可以找到注入的`<style>`标签，它跟我们在`styles`参数中定义的完全一样：

```
.highlight {
  border: 2px dashed red;
  text-align: center;
  margin-bottom: 20px;
}
```

使用`ViewEncapsulation.None`的缺点是，因为没有进行任何封装，所以它的样式会影响到其他组件。在图13-8中可以看到，使用`ViewEncapsulation.Native`的组件已经受到了这个新组件的样式的影响。但有时候这可能恰恰是你想要的。

你可以注释掉`StyleSampleApp`模板中的`<no-encapsulation></no-encapsulation>`这行代码来看一看区别。

14.2　创建 popup 指令：引用并修改宿主元素

宿主元素是指令或组件被绑定到的元素。有时组件可能需要往它的宿主元素上附加一些标记或行为。

在这个示例中，我们会创建一个popup指令。它会往宿主元素上附加行为，在宿主元素被点击时显示一条信息。



组件与指令：两者的区别是什么？

组件和指令有着密不可分的关系，但它们略有不同。

你或许曾听说过“组件就是有视图的指令”。其实这并不完全正确。组件自带的功能使它很容易添加视图，但指令同样也可以有视图。事实上，**组件是用指令来实现的**。

一个很好的例子就是`ngIf`，它根据条件来渲染视图。

但我们可以使用**指令在没有模板**的情况下给元素附加行为。

你可以这样认为：组件就是指令，但组件必须有视图。指令可以有视图，也可以没有。

如果你选择在指令中渲染视图（模板）的话，可以对该模板的呈现方式进行更多的控制。在本章的后面我们会讨论如何对模板进行控制。

14.2.1　popup 指令的结构

现在来编写我们的首个指令。我们希望在点击一个带有popup属性的DOM元素时，该指令能显示出一个提示消息。这个消息是通过该元素的message属性来指定的。

我们希望它看起来如下所示：

```
<element popup message="Some message"></element>
```

为了让这个组件正常工作，我们还要做一些事：

- 接收来自宿主元素的message属性；
- 当宿主元素被点击时得到通知。

我们这就开始编写它。

code/advanced_components/app/ts/host/steps/host_01.ts

```
@Directive({
  selector: '[popup]'
})
class Popup {
  constructor() {
    console.log('Directive bound');
```

```
  }
}
```

我们使用Directive注解并将selector参数设置为[popup]。这可以让该指令绑定到任何定义了popup属性的元素。

现在来创建一个应用，它包含一个有popup属性的元素。

code/advanced_components/app/ts/host/steps/host_01.ts

```
@Component({
  selector: 'host-sample-app',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>
  `
})
export class HostSampleApp1 {
}
```

运行这个应用时，我们期望Directive bound消息会被打印到控制台中，这表示我们已经成功绑定了模板中的第一个<div>（如图14-9所示）。

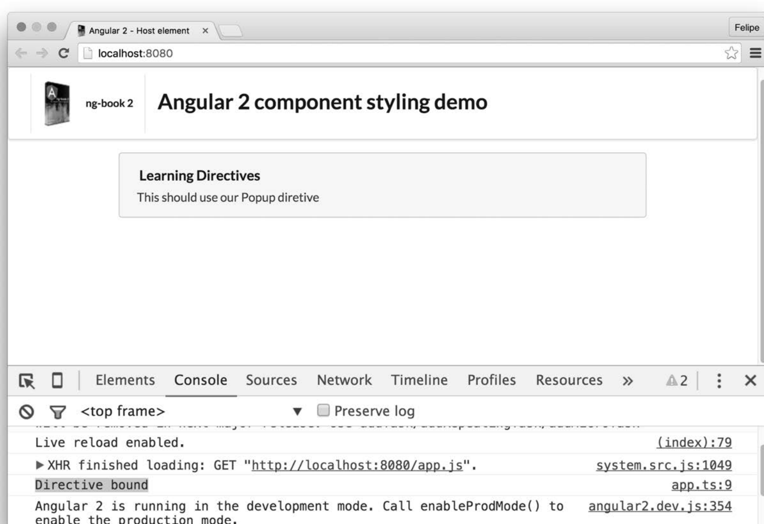


图14-9　绑定到宿主元素

14.2.2 使用 ElementRef

如果我们想对指令所绑定的宿主元素进行更多控制，可以使用内置的ElementRef类。

这个类保存着指定Angular元素的相关信息，使用它的nativeElement属性可以获取原生的DOM元素。

为了看到指令所绑定的元素，我们可以在构造函数中接收ElementRef并把它打印到控制台中。

code/advanced_components/app/ts/host/steps/host_02.ts

```
@Directive({
  selector: '[popup]'
})
class Popup {
  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }
}
```

我们还可以往页面中添加另一个元素，它也使用这个指令。这样就可以看见控制台中打印了两个不同的ElementRef。

code/advanced_components/app/ts/host/steps/host_02.ts

```
@Component({
  selector: 'host-sample-app',
  template: `
  <div class="ui message" popup>
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup></i>
  `
})
export class HostSampleApp2 {
}
```

现在，当运行应用时，可以看到两个不同的ElementRef：一个是div.ui.message，另一个是i.alarm.icon。这表示该指令已经成功绑定了两个不同的宿主元素，如图14-10所示。

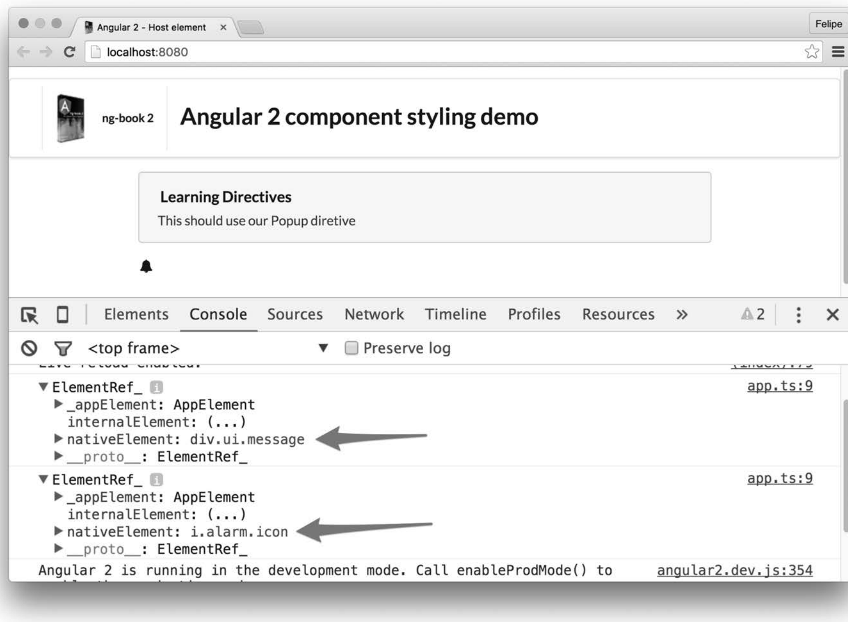


图14-10　两个ElementRef

14.2.3　绑定到 host 属性

我们的下一个目标是在宿主元素被点击时做一些事。

我们之前学过，在Angular中给元素绑定事件的方法是使用(event)语法。

为了给宿主元素绑定事件，我们必须做一些类似的事情，不同之处是这次使用指令的host属性。host属性允许指令改变其宿主元素的属性和行为。

我们还希望宿主元素使用它的message属性来定义点击时要弹出的消息。

首先，在指令中添加inputs属性。我们导入Input，并使用@Input注解来修饰这个输入属性。

```
import { Component, Input } from '@angular/core';
...
class Popup {
  @Input() message: String;
  ...
}
```

这段代码表示我们有一个名为message的属性，并且期望接收一个与之同名的输入。

接着，我们通过往@Component注解上添加host属性来把它绑定到宿主元素上。

code/advanced_components/app/ts/host/steps/host_03.ts

```
@Directive({
  selector: '[popup]',
  host: {
    '(click)': 'displayMessage()'
  }
})
```

然后，当宿主元素被点击时就会调用指令的displayMessage方法，它会显示宿主元素定义的消息。

现在代码如下所示。

code/advanced_components/app/ts/host/steps/host_03.ts

```
class Popup {
  @Input()message: String;

  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }

  displayMessage(): void {
    alert(this.message);
  }
}
```

最后，我们需要修改应用的模板，为每个元素添加要显示的消息。

code/advanced_components/app/ts/host/steps/host_03.ts

```
@Component({
  selector: 'host-sample-app',
  template: `
  <div class="ui message" popup
       message="Clicked the message">
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup
     message="Clicked the alarm icon"></i>
  `
})
export class HostSampleApp3 {
}
```

注意，这里使用了两次popup指令并传入了不同的message属性。这意味着当我们运行本应

14

用时，点击信息内容或者图标将会看到不同的弹出信息，分别如图14-11和图14-12所示。

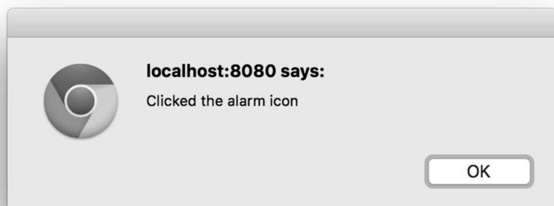


图14-11　弹出信息1

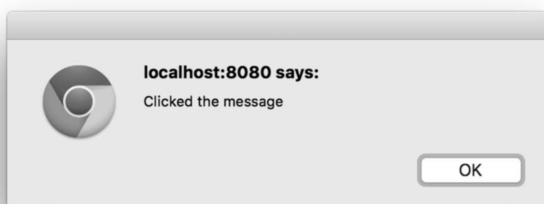


图14-12　弹出信息2

14.2.4　添加按钮并使用 exportAs

假设现在又来了新需求：通过点击按钮来手动触发弹出信息。那么该如何在宿主元素之外触发弹出信息呢？

为了实现这个目标，我们要让指令在模板中的任何地方都能被访问到。正如我们在之前章节中讨论过的，可以使用模板变量来引用组件。我们也可以用同样的方式来引用指令。

为了可以在模板中引用指令，就要使用exportAt属性。这将允许宿主元素（或宿主元素的子元素）使用#var="exportName"语法定义一个模板变量来引用指令。

让我们把exportAs属性添加到指令中。

code/advanced_components/app/ts/host/steps/host_04.ts

```
@Directive({
  selector: '[popup]',
  exportAs: 'popup',
  host: {
    '(click)': 'displayMessage()'
  }
})
class Popup {
```

```
  @Input() message: String;

  constructor(_elementRef: ElementRef) {
    console.log(_elementRef);
  }

  displayMessage(): void {
    alert(this.message);
  }
}
```

现在我们需要修改这两个元素来导出模板变量。

code/advanced_components/app/ts/host/steps/host_04.ts

```
  template: `
  <div class="ui message" popup #popup1="popup"
       message="Clicked the message">
    <div class="header">
      Learning Directives
    </div>

    <p>
      This should use our Popup diretive
    </p>
  </div>

  <i class="alarm icon" popup #p2="popup"
     message="Clicked the alarm icon"></i>
```

可以看到，我们用模板变量#p1代表div.message，用#p2代表icon。

现在再添加两个按钮，分别触发它们的弹出信息。

code/advanced_components/app/ts/host/steps/host_04.ts

```
  <div style="margin-top: 20px;">
    <button (click)="popup1.displayMessage()" class="ui button">
      Display popup for message element
    </button>

    <button (click)="p2.displayMessage()" class="ui button">
      Display popup for alarm icon
    </button>
  </div>
```

现在刷新页面并分别点击每个按钮，每条消息都会如预期那样出现。

14.3 使用内容投影创建消息面板

有时，我们在创建组件的时候想要把组件内部的标记作为一个参数传给组件。这种技术就叫作内容投影（content projection）。它能让我们指定一些会扩散到更大模板之中的标记。



在AngularJS中，这种技术被称为**透传**（transclusion）。

我们来创建一个指令，它将渲染一个比较好看的消息，如图14-13所示。

Learning Directives

This should use our Popup diretive

图14-13　popup指令渲染的消息

我们的最终目标是写如下标记。

```
<div message header="My Message">
  This is the content of the message
</div>
```

它将被渲染成更复杂的HTML。

```
<div class="ui message">
  <div class="header">
    My Message
  </div>

  <p>
    This is the content of the message
  </p>
</div>
```

这里面临两个挑战：我们要给宿主元素添加两个CSS类（`ui`和`message`），还要把`div`中的内容添加到标记中的一个指定位置。

14.3.1　改变 host 属性的 CSS 类

和之前添加事件一样，为了给宿主元素添加属性，要使用`host`属性；但是在这里我们定义了属性的名称和值，而不是使用`(event)`的写法。在这个例子中是这样的。

```
host: { 'class': 'ui message' }
```

它会修改宿主元素，把这些类添加到`class`属性中。

14.3.2　使用 ng-content

下一个挑战是将宿主元素节点原来的子节点包含进视图中的指定部分。要做到这一点，我们使用`ng-content`指令。

因为这个指令需要模板，所以在这里改用组件，并编写如下代码。

code/advanced_components/app/ts/content-projection/content-projection.ts

```
@Component({
  selector: '[message]',
  host: {
    'class': 'ui message'
  },
  template: `
    <div class="header">
      {{ header }}
    </div>
    <p>
      <ng-content></ng-content>
    </p>
  `
})
export class Message {
  @Input() header: string;

  ngOnInit(): void {
    console.log('header', this.header);
  }
}
```

下面是一些要点：

- 用`@inputs`注解表明我们希望接收宿主元素上设置的`header`属性；
- 用组件的`host`属性把宿主元素的`class`属性设置为`ui message`；
- 使用`<ng-content></ng-content>`将宿主元素的子节点投影到模板中的指定位置。

当我们在浏览器中打开应用并检查有`message`属性的`div`时，会看到它正如我们所预期的那样工作，如图14-14所示。

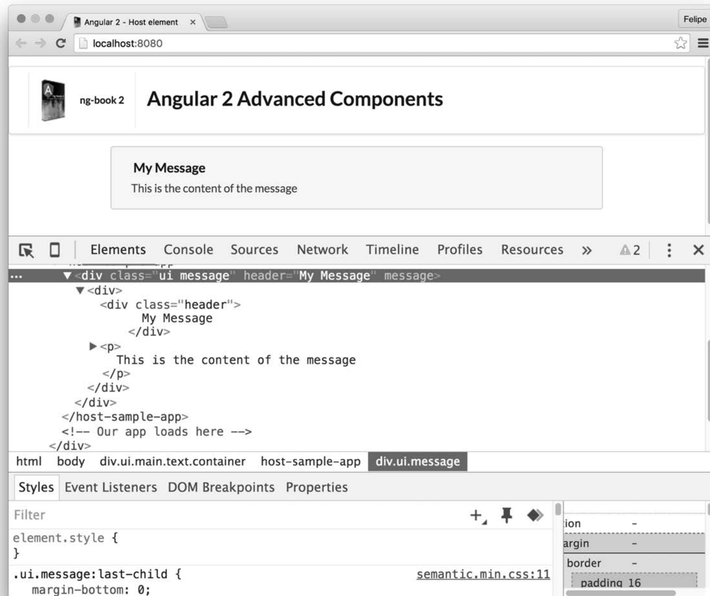


图14-14　投影进来的内容

14.4　查询相邻的指令：编写标签页

如果你能创建一个完全封装了自身行为的组件，那当然很棒。

然而，随着组件功能的不断扩展，将组件切割成一些更小的组件再将它们组合在一起就变得有意义了。

一个拥有多个标签页的标签面板是组件协同工作的好例子。标签面板或者标签集合是由多个标签页组合而成的。在这个场景中，我们有一个父组件（标签集合）和多个子组件（标签页）。单独看标签面板或标签页没有意义，但把所有逻辑都放在同一个组件中又太笨重了。因此，我们将在这个示例中讲解如何让这些单独的组件协同工作。

下面来编写这些组件，最终目标是这样用。

```
<tabset>
  <tab title="Tab 1">Tab 1</tab>
  <tab title="Tab 2">Tab 2</tab>
  ...
</tabset>
```

我们将使用Semantic UI的Tab组件[1]来渲染标签页。

① http://semantic-ui.com/modules/tab.html#/examples

14.4.1 Tab 组件

先来编写Tab组件。

code/advanced_components/app/ts/tabs/tabs.ts

```
@Component({
  selector: 'tab',
  template: `
  <div class="ui bottom attached tab segment"
       [class.active]="active">

    <ng-content></ng-content>

  </div>
  `
})
class Tab {
  @Input() title: string;
  active: boolean = false;
  name: string;
}
```

这里没有什么新概念。我们声明了一个组件，它的选择器是tab并且接收一个输入属性title。

然后渲染一个<div>标签，并使用前一节中学过的内容投影概念把<tab>指令的行内内容嵌入这个div。

接下来声明三个组件属性：title、active和name。需要注意的是，我们把title属性添加到了@Input('title')注解中。这个注解告诉Angular自动把输入属性title和组件属性title进行绑定。

14.4.2 Tabset 组件

现在让我们转向Tabset组件，用它来包裹住标签页。

code/advanced_components/app/ts/tabs/tabs.ts

```
@Component({
  selector: 'tabset',
  template: `
  <div class="ui top attached tabular menu">
    <a *ngFor="let tab of tabs"
        class="item"
        [class.active]="tab.active"
        (click)="setActive(tab)">

      {{ tab.title }}

    </a>
  </div>
```

```
    <ng-content></ng-content>
    `
})
class Tabset implements AfterContentInit {
  @ContentChildren(Tab) tabs: QueryList<Tab>;

  constructor() {
  }

  ngAfterContentInit() {
    this.tabs.toArray()[0].active = true;
  }

  setActive(tab: Tab) {
    this.tabs.toArray().forEach((t) => t.active = false);
    tab.active = true;
  }
}
```

我们来分别讲解它的实现，这样可以更好地学习它引入的新概念。

1. Tabset的@Component注解

@Component部分没有什么新概念。我们使用<tabset>作为选择器。

模板本身使用ngFor来遍历tabs属性。如果一个tab的active标记是true，那么它就会在用来渲染tab的<a>元素上添加CSS类active。

我们还指定了，在初始化div之后、在ng-content所在位置渲染所有标签。

2. Tabset类

现在让我们把注意力转向Tabset类。这里的第一个新概念就是Tabset类实现了AfterContentInit。这个生命周期钩子告诉Angular，一旦子组件的内容初始化，就调用类的方法（ngAfterContentInit）。

3. Tabset的ContentChildren和QueryList

接下来，我们声明tabs属性，用它来保存在tabset中声明的每个Tab组件。注意，这里声明的不是一个Tab的数组，而是使用QueryList类并传入泛型Tab。这是为什么呢？

QueryList是Angular提供的类。当我们同时使用QueryList和ContentChildren时，Angular就会将匹配查询的组件填充到QueryList，然后在应用状态变更时保持这些填充项的更新。

然而，QueryList需要ContentChildren来进行填充。我们这就来看一看。

在tab实例对象上，我们添加了@ContentChildren(Tab)注解。这个注解告诉Angular要在tabs参数中注入所有Tab类型的直接子指令。然后再将其赋值给组件的tabs属性。有了tabs属性，我们就可以获得并使用所有的子Tab组件了。

4. 初始化Tabset

当这个组件初始化之后，我们希望它的第一个标签页是激活的。为了做到这点，要使用ngAfterContentInit函数（AfterContentInit钩子对应的实现方法）。注意这里使用this.tabs.toArray()将Angular的QueryList强制转换为原生的TypeScript数组。

5. Tabset的setActive方法

最后，我们定义了setActive方法。当点击模板中的标签页时就会调用这个方法，例如(click)="setActive(tab)"。这个函数会遍历所有标签页，将它们的active属性设置为false。然后把我们点击的标签页设置为激活页。

14.4.3　使用 Tabset

下一个任务是开发一个应用组件，它将使用我们创建好的这两个组件。我们可以这样做。

code/advanced_components/app/ts/tabs/tabs.ts

```
@Component({
  selector: 'tabs-sample-app',
  template: `
  <tabset>
    <tab title="First tab">
      Lorem ipsum dolor sit amet, consectetur adipisicing elit.
      Quibusdam magni quia ut harum facilis, ullam deleniti porro
      dignissimos quasi at molestiae sapiente natus, neque voluptatum
      ad consequuntur cupiditate nemo sunt.
    </tab>
    <tab *ngFor="let tab of tabs" [title]="tab.title">
      {{ tab.content }}
    </tab>
  </tabset>
  `
})
export class TabsSampleApp {
  tabs: any;

  constructor() {
    this.tabs = [
      { title: 'About', content: 'This is the About tab' },
      { title: 'Blog', content: 'This is our blog' },
      { title: 'Contact us', content: 'Contact us here' },
    ];
  }
}
```

我们使用tabs-sample-app作为组件的选择器并且使用了组件Tabset和Tab。

我们在模板中创建了一个tabset组件并添加了一个静态的tab组件（第一页），然后往组件控制器类中的tabs属性中又添加了几个tab组件，阐明了动态渲染tab组件的方法。完成后的应用如图14-15所示。

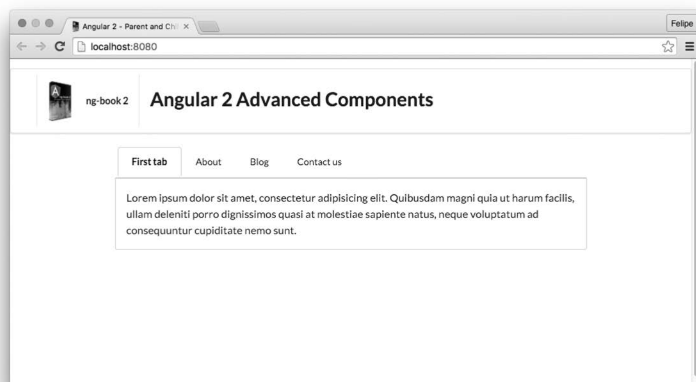


图14-15　使用Tabset的应用

14.5　生命周期钩子

Angular提供了一些生命周期钩子。在指令生命周期的每个阶段之前或之后，它们允许你添加并执行一些代码。

Angular提供的生命周期钩子如下：

- OnInit
- OnDestroy
- DoCheck
- OnChanges
- AfterContentInit
- AfterContentChecked
- AfterViewInit
- AfterViewChecked

这些钩子的使用方法遵循相似的模式。

为了得到这些事件的通知，你需要：

(1) 声明你的指令类实现接口；

(2) 声明钩子对应的ng方法（例如，ngOnInit）。

每个方法名都以ng开头，再加上钩子的名字。比如，OnInit要声明ngOnInit方法，AfterContentInit要声明ngAfterContentInit方法，以此类推。

当Angular知道组件实现了这些函数后，就会在适当的时机调用它们。

下面分别看看每个钩子的用法以及使用场景。



实际上，让这个类实现（implement）该接口并不是必需的，也可以只创建此钩子要求的方法。不过实现该接口是一项最佳实践[1]，它能在强类型和编辑器等方面给你带来好处。

14.5.1 OnInit 和 OnDestroy

在指令的属性初始化完成之后、子指令的属性开始初始化之前，Angular会调用OnInit钩子。

同样，在指令的实例销毁之前，Angular调用OnDestroy钩子。它最典型的应用场景是，当指令销毁、要做一些清理工作时。

为了说明这些，我们来编写一个同时实现了OnInit和OnDestroy的组件。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_01.ts

```
@Component({
  selector: 'on-init',
  template: `
  <div class="ui label">
    <i class="cubes icon"></i> Init/Destroy
  </div>
  `
})
class OnInitCmp implements OnInit, OnDestroy {
  ngOnInit(): void {
    console.log('On init');
  }

  ngOnDestroy(): void {
    console.log('On destroy');
  }
}
```

在这个组件中，当钩子被调用时，我们只是往控制台中打印字符串On init和On destroy。

要测试这些钩子，我们就要在应用组件中使用这些组件，并用ngIf来根据布尔值决定是否显示我们的组件。然后添加一个按钮让我们切换这个布尔型标志：当标记变为假时，组件会从页面中移除，OnDestroy就会被调用；当标记变为真时，OnInit钩子会被调用。

应用组件看起来如下所示。

14

① https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_01.ts

```
@Component({
  selector: 'lifecycle-sample-app',
  template: `
  <h4 class="ui horizontal divider header">
    OnInit and OnDestroy
  </h4>

  <button class="ui primary button" (click)="toggle()">
    Toggle
  </button>
  <on-init *ngIf="display"></on-init>
  `
})
export class LifecycleSampleApp1 {
  display: boolean;

  constructor() {
    this.display = true;
  }

  toggle(): void {
    this.display = !this.display;
  }
}
```

首次运行该应用时，可以看到OnInit钩子在组件首次初始化后被调用了，如图14-16所示。

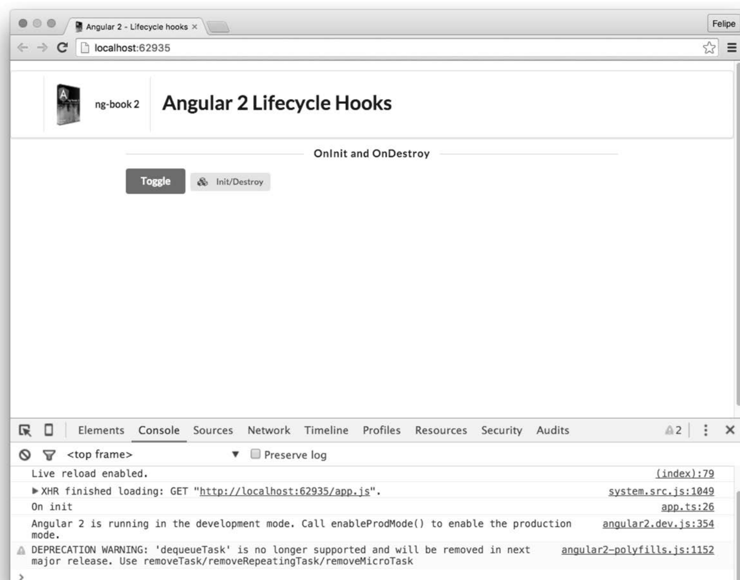


图14-16　组件的初始状态

在第一次点击Toggle按钮时，组件被销毁，OnDestroy钩子也如预期一般被调用了，如图14-17所示。

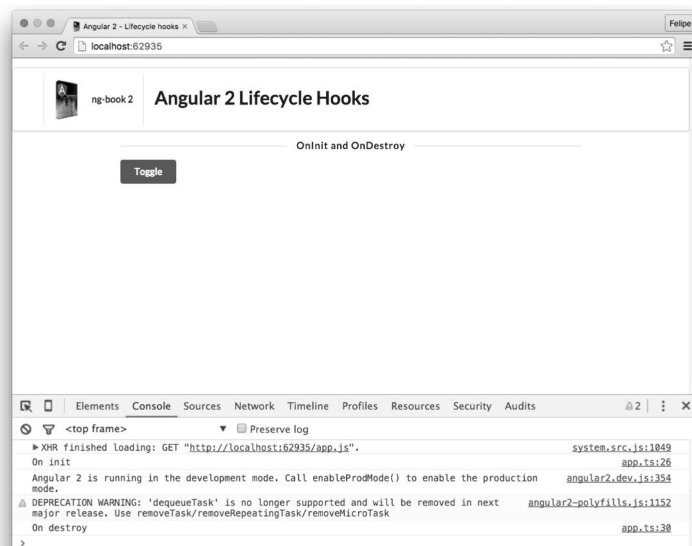


图14-17　OnDestroy钩子：首次点击Toggle按钮

如果再次点击Toggle按钮，结果将如图14-18所示。

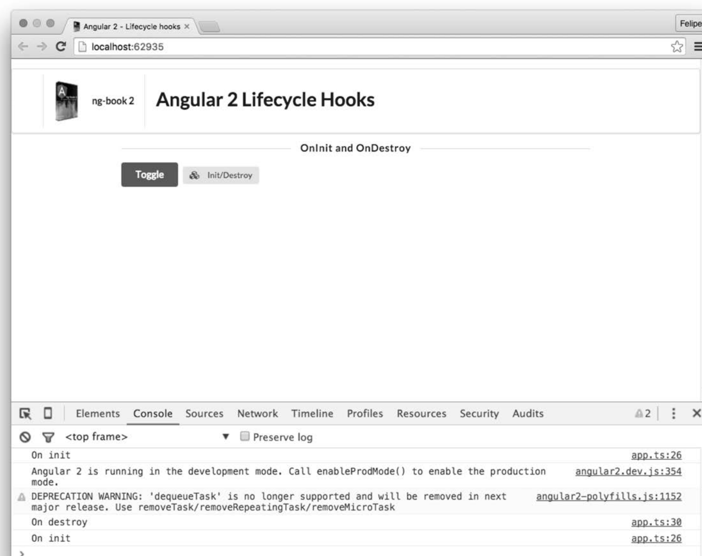


图14-18　OnDestroy钩子：再次点击Toggle按钮

14.5.2　OnChanges

OnChanges钩子在一个或多个组件属性更改后调用。ngOnChanges方法会接收一个参数来告诉你哪些属性发生了改变。

为了更好地理解这一点，我们来编写一个评论组件。该组件有两个输入属性：name和comment。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
@Component({
  selector: 'on-change',
  template: `
  <div class="ui comments">
    <div class="comment">
      <a class="avatar">
        <img src="app/images/avatars/matt.jpg">
      </a>
      <div class="content">
        <a class="author">{{name}}</a>
        <div class="text">
          {{comment}}
        </div>
      </div>
    </div>
  </div>
  `
})
class OnChangeCmp implements OnChanges {
  @Input('name') name: string;
  @Input('comment') comment: string;

  ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
    console.log('Changes', changes);
  }
}
```

最重要的一点是，这个组件实现了OnChanges接口，并声明了该接口的ngOnChanges方法。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
  ngOnChanges(changes: {[propName: string]: SimpleChange}): void {
    console.log('Changes', changes);
  }
```

当name属性或comment属性的值发生变化时，这个方法就会被触发。这时，我们就会收到一个对象，它把发生变化的字段映射到SimpleChange对象中。

每个SimpleChange实例都有两个字段：currentValue和previousValue。如果组件的name和comment属性都发生了变化，那么该方法的changes值就应该是这样的。

```
{
  name: {
    currentValue: 'new name value',
```

```
    previousValue: 'old name value'
  },
  comment: {
    currentValue: 'new comment value',
    previousValue: 'old comment value'
  }
}
```

现在对应用组件进行修改，让它使用我们的组件并添加一个小型表单。这样就可以试试与组件的name属性和comment属性的交互了。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
@Component({
  selector: 'lifecycle-sample-app',
  template: `
  <h4 class="ui horizontal divider header">
    OnInit and OnDestroy
  </h4>

  <button class="ui primary button" (click)="toggle()">
    Toggle
  </button>
  <on-init *ngIf="display"></on-init>

  <h4 class="ui horizontal divider header">
    OnChange
  </h4>

  <div class="ui form">
    <div class="field">
      <label>Name</label>
      <input type="text" #namefld value="{{name}}"
             (keyup)="setValues(namefld, commentfld)">
    </div>

    <div class="field">
      <label>Comment</label>
      <textarea (keyup)="setValues(namefld, commentfld)"
                rows="2" #commentfld>{{comment}}</textarea>
    </div>
  </div>

  <on-change [name]="name" [comment]="comment"></on-change>
  `
})
export class LifecycleSampleApp2 {
  display: boolean;
  name: string;
  comment: string;

  constructor() {
    this.display = true;
    this.name = 'Felipe Coury';
```

```
    this.comment = 'I am learning so much!';
  }

  setValues(namefld, commentfld): void {
    this.name = namefld.value;
    this.comment = commentfld.value;
  }

  toggle(): void {
    this.display = !this.display;
  }
}
```

重点是我们往模板中添加了一个新的表单，这个表单有name和comment两个字段。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
<div class="ui form">
  <div class="field">
    <label>Name</label>
    <input type="text" #namefld value="{{name}}"
           (keyup)="setValues(namefld, commentfld)">
  </div>

  <div class="field">
    <label>Comment</label>
    <textarea (keyup)="setValues(namefld, commentfld)"
              rows="2" #commentfld>{{comment}}</textarea>
  </div>
</div>
```

无论在name字段还是comment字段的keyup事件触发时，我们都通过模板变量调用setValues方法。模板变量namefld和commentfld分别代表这里的input和textarea。

这个方法只是取出这些字段的值并更新对应的name和comment属性。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_02.ts

```
setValues(namefld, commentfld): void {
  this.name = namefld.value;
  this.comment = commentfld.value;
}
```

现在，当我们第一次打开应用时，就会看到OnChanges钩子被调用了，如图14-19所示。

这个事件在刚刚设置了初始值时发生在LifecycleSampleApp组件的构造函数中。

如果在Name输入框中进行输入，就会看到钩子函数不断被重复调用。如图14-20所示，当我们在Name输入框中粘贴Nate Murray时，控制台正如我们预期的那样反映出了值的变化。

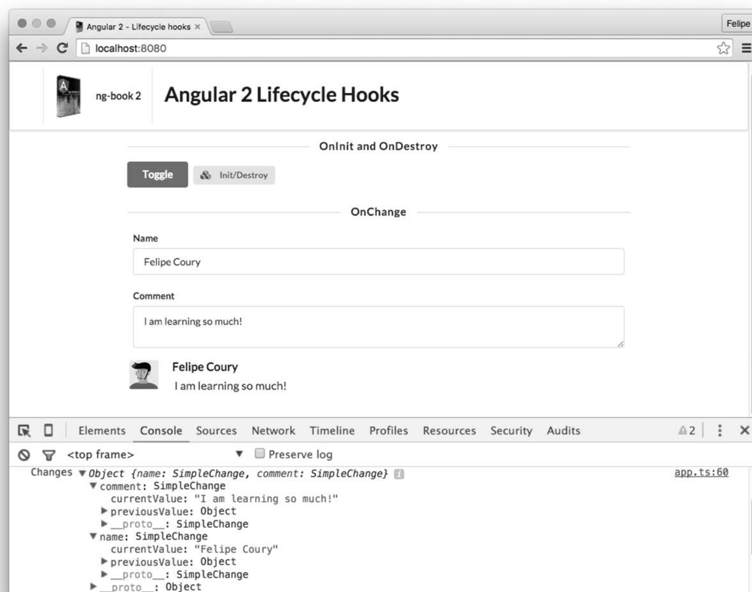


图14-19　OnChanges钩子：首次打开应用时

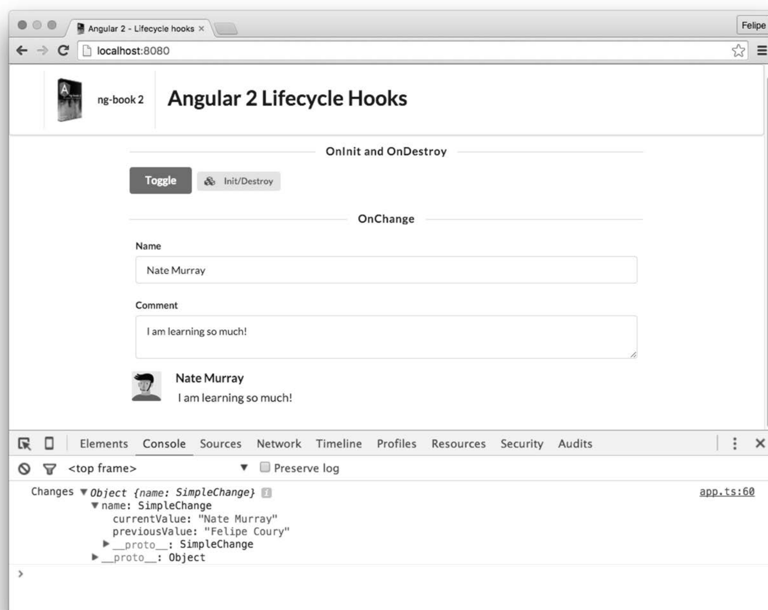


图14-20　OnChanges钩子：输入后

14.5.3　DoCheck

Angular默认的通知系统就是通过OnChanges实现的，每当Angular的变更检测机制检测到指令的属性变化时就会触发它。

然而，有时候这种变更通知机制可能开销过大，尤其是在对性能要求较高的场景下。

有时候，我们只想在特定的条件下进行一些操作，比如在移除或添加一个项目时，或是在某个特定的属性发生变化时。

如果遇到上述场景之一，就可以使用DoCheck钩子。



有一点非常重要，如果我们同时实现了OnChanges和DoCheck，那么OnChanges会被DoCheck覆盖，也就是说OnChanges会被忽略。

1. 变更检测

为了找出有哪些变化，Angular提供了differ（差分器）类。differ会对指令的某个属性进行计算，以确定它是否发生了改变。

有两种内置的differ类型：迭代differ和键值对differ。

2. 迭代differ

当我们使用列表类的数据结构并且只想知道在列表中添加或删除了哪些条目时，应该使用迭代differ。

3. 键值对differ

当我们使用字典类数据结构时，应该使用键值对differ；它在键一级工作。这个differ会识别出键的添加、删除或某个键对应值的改变。

4. 使用`do-check-item`渲染单条评论

为了阐明这些概念，我们来构建一个渲染一系列评论的组件，如图14-21所示。

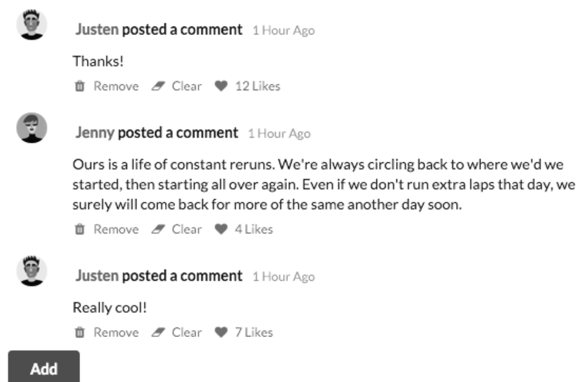


图14-21　DoCheck钩子示例

首先，我们编写一个渲染单条评论的组件。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
@Component({
  selector: 'do-check-item',
  outputs: ['onRemove'],
  template: `
  <div class="ui feed">
    <div class="event">
      <div class="label" *ngIf="comment.author">
        <img src="/app/images/avatars/{{comment.author.toLowerCase()}}.jpg">
      </div>
      <div class="content">
        <div class="summary">
          <a class="user">
            {{comment.author}}
          </a> posted a comment
          <div class="date">
            1 Hour Ago
          </div>
        </div>
        <div class="extra text">
          {{comment.comment}}
        </div>
        <div class="meta">
          <a class="trash" (click)="remove()">
            <i class="trash icon"></i> Remove
          </a>
          <a class="trash" (click)="clear()">
            <i class="eraser icon"></i> Clear
          </a>
          <a class="like" (click)="like()">
            <i class="like icon"></i> {{comment.likes}} Likes
          </a>
        </div>
```

```
      </div>
    </div>
  </div>
  `
})
```

我们声明了组件的元数据。组件会接收输入属性comment并渲染它，还会在点击删除按钮时发出一个事件。

继续看组件类的实现。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
class DoCheckItem implements DoCheck {
  @Input('comment') comment: any;
  onRemove: EventEmitter<any>;
  differ: any;
```

在这个类声明中，我们实现了DoCheck接口，并且声明了输入属性comment、输出事件onRemove和一个differ属性。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
  constructor(differs: KeyValueDiffers) {
    this.differ = differs.find([]).create(null);
    this.onRemove = new EventEmitter();
  }
```

在这个构造函数中，我们用differs变量接收了一个KeyValueDiffers的实例，然后通过differs.find([]).create(null)语法创建了一个键值对differ的实例。我们还初始化了事件发射器onRemove。

接下来，我们来实现接口要求的ngDoCheck方法。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
  ngDoCheck(): void {
    var changes = this.differ.diff(this.comment);

    if (changes) {
      changes.forEachAddedItem(r => this.logChange('added', r));
      changes.forEachRemovedItem(r => this.logChange('removed', r));
      changes.forEachChangedItem(r => this.logChange('changed', r));
    }
  }
```

这里用键值对differ检测了变更，只要调用diff方法并提供想要检查的属性就可以了。在这个例子中，我们想知道comment属性是否发生了变化。

当没有检测到任何变化时，返回值就是null。如果有变化，我们可以调用differ上的三个不同的迭代方法：

- forEachAddedItem，用于枚举所有新增的键；

- forEachRemovedItem，用于枚举所有删除的键；
- forEachChangedItem，用于枚举所有变化的键。

每个方法都会调用我们提供的接收record参数的回调函数。对于键值对differ，这个record参数是KVChangeRecord类的实例（如图14-22所示）。

```
▼KVChangeRecord {key: "likes", previousValue: null, currentValue: 10, _nextPrevious: null, _next: null…}
   _next: null
   _nextAdded: null
   _nextChanged: null
   _nextPrevious: null
   _nextRemoved: null
   _prevRemoved: null
   currentValue: 10
   key: "likes"
   previousValue: 10
```

图14-22 KVChangeRecord实例的一个例子

用来了解变化的最重要的几个字段是key、previousValue和currentValue。

接下来，我们写一个方法，把发生的这些变化以通俗易懂的句子输出到控制台中。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
logChange(action, r) {
  if (action === 'changed') {
    console.log(r.key, action, 'from', r.previousValue, 'to', r.currentValue);
  }
  if (action === 'added') {
    console.log(action, r.key, 'with', r.currentValue);
  }
  if (action === 'removed') {
    console.log(action, r.key, '(was ' + r.previousValue + ')');
  }
}
```

最后，我们来写几个方法，帮助我们改变组件中的值以便触发DoCheck钩子。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
remove(): void {
  this.onRemove.emit(this.comment);
}

clear(): void {
  delete this.comment.comment;
}

like(): void {
  this.comment.likes += 1;
}
```

remove()方法会发出事件，表示用户请求删除这条评论。clear()方法会把评论文字从评论对象中删除。like()方法会增加这条评论的“赞”数。

5. 使用do-check渲染评论列表

写好了表示单条评论的组件之后，我们再来写第二个组件，它负责渲染评论列表。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
@Component({
  selector: 'do-check',
  template: `
  <do-check-item [comment]="comment"
    *ngFor="let comment of comments" (onRemove)="removeComment($event)">
  </do-check-item>

  <button class="ui primary button" (click)="addComment()">
    Add
  </button>
  `
})
```

组件的元数据十分简单：使用上面创建的组件，然后用ngFor来遍历组件列表并渲染它们。我们还加了一个按钮让用户添加新评论。

接下来实现评论列表类DoCheckCmp。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
class DoCheckCmp implements DoCheck {
  comments: any[];
  iterable: boolean;
  authors: string[];
  texts: string[];
  differ: any;
```

我们声明了要用的变量：comments、iterable、authors和texts。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
  constructor(differs: IterableDiffers) {
    this.differ = differs.find([]).create(null);
    this.comments = [];

    this.authors = ['Elliot', 'Helen', 'Jenny', 'Joe', 'Justen', 'Matt'];
    this.texts = [
      "Ours is a life of constant reruns. We're always circling back to where we\
'd we started, then starting all over again. Even if we don't run extra laps tha\
t day, we surely will come back for more of the same another day soon.",
      'Really cool!',
      'Thanks!'
    ];

    this.addComment();
  }
```

对于这个组件，我们使用了迭代differ。可以看到这里用来创建differ的类是IterableDiffers，但创建differ的方式还是和以前一样。

在构造函数中,我们还初始化了作者列表和评论文字列表,会在添加新评论的时候用到它们。

最后，我们调用addComment()方法。这样，评论列表在应用刚刚初始化时不会是空白的。

接下来的三个方法是用来添加一条新评论的。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
getRandomInt(max: number): number {
  return Math.floor(Math.random() * (max + 1));
}

getRandomItem(array: string[]): string {
  let pos: number = this.getRandomInt(array.length - 1);
  return array[pos];
}

addComment(): void {
  this.comments.push({
    author: this.getRandomItem(this.authors),
    comment: this.getRandomItem(this.texts),
    likes: this.getRandomInt(20)
  });
}

removeComment(comment) {
  let pos = this.comments.indexOf(comment);
  this.comments.splice(pos, 1);
}
```

我们声明了两个方法，它们分别返回一个随机数和一个数组中的随机项。

最后，addComment()方法会使用随机作者、随机文本和随机点赞数来添加一条新评论。

接下来是removeComment()方法，它用来从列表中删除一条评论。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
removeComment(comment) {
  let pos = this.comments.indexOf(comment);
  this.comments.splice(pos, 1);
}
```

最后声明变更检测方法ngDoCheck()。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_03.ts

```
ngDoCheck(): void {
  var changes = this.differ.diff(this.comments);

  if (changes) {
    changes.forEachAddedItem(r => console.log('Added', r.item));
    changes.forEachRemovedItem(r => console.log('Removed', r.item));
  }
}
```

尽管在行为上与键值对differ一样，但是迭代differ只提供了添加和删除条目的方法。

当运行应用时，我们得到了一个只有一条评论的列表（如图14-23所示）。

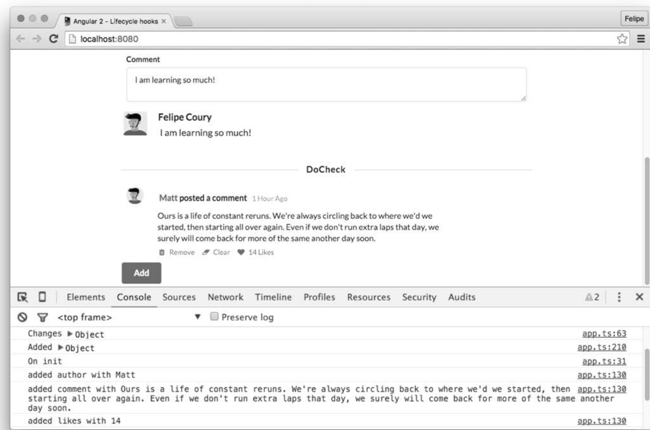


图14-23　初始状态

我们还看到一些信息被打印到了控制台中，就像下面这样。

```
added author with Matt
...
added likes with 14
```

我们来看看，点击Add按钮来添加一条新评论时会发生什么（如图14-24所示）。

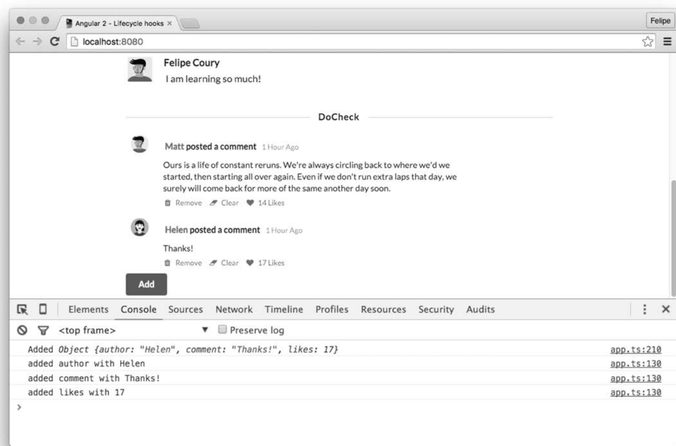


图14-24　添加的评论

可以看到迭代differ识别出了添加到列表中的新评论对象`{author: "Hellen", comment:`

"Thanks!", likes: 17}。

评论对象中单独的属性变化也打印出来了，也就是键值对differ检测到的。

```
added author with Helen
added comment with Thanks!
added likes with 17
```

现在点击这条新评论的Likes图标（如图14-25所示）。

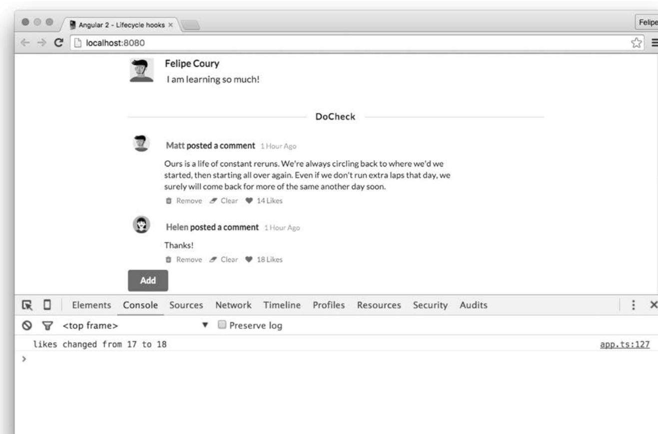


图14-25　点赞数变化

现在只有like属性的变化会被检测到。

如果点击Clear图标，它会从评论对象中删除comment键（如图14-26所示）。

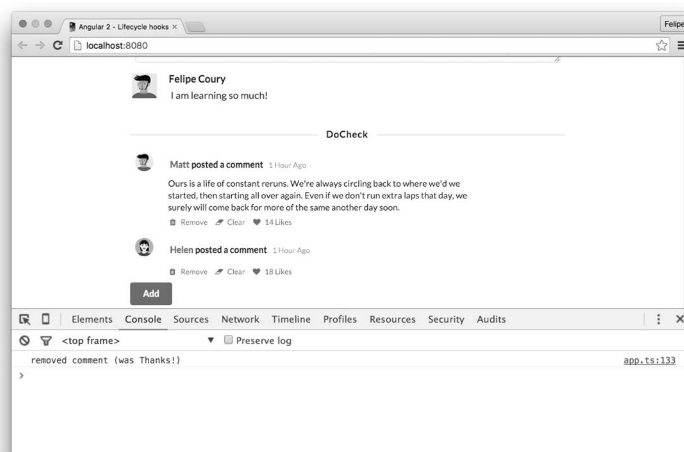


图14-26　清空评论内容

打印出的日志证实这个键确实被删除了。

最后，我们通过点击Remove图标删除最后一条评论（如图14-27所示）。

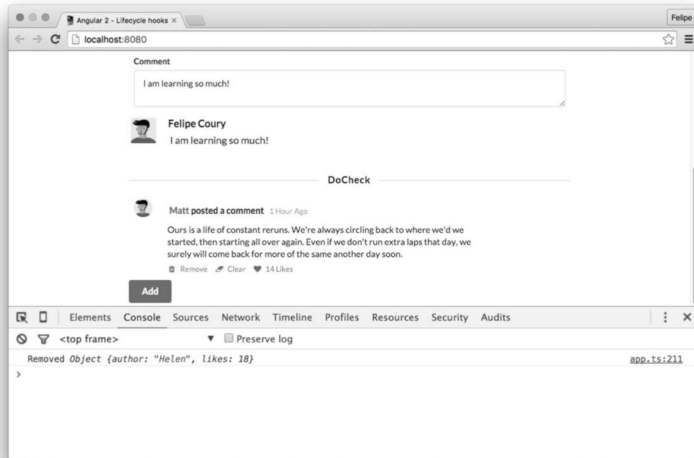


图14-27　删除评论

如预期一样，我们得到了一条对象被删除的日志。

14.5.4　AfterContentInit、AfterViewInit、AfterContentChecked 和 AfterViewChecked

AfterContentInit钩子的调用发生在OnInit之后。一旦组件或指令的内容初始化完成，就会立即调用它。

AfterContentChecked也类似，不过它是在指令检查结束后调用的。这里的“检查”是指变更检测系统进行的检查。

另外两个钩子AfterViewInit和AfterViewChecked会紧跟着上述内容钩子，在视图完全初始化之后触发。但是这两个钩子只适用于组件，不能用于指令。

同时，AfterXXXInit之类的钩子在整个指令生命周期里都只会被调用一次，而AfterXXXChecked之类的钩子在每次变更检测周期后都会被调用。

为了更好地理解这些，我们来编写另一个组件，它会对每个生命周期钩子都打印日志到控制台。它还有一个counter属性，可以通过点击按钮来增加计数。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```
@Component({
  selector: 'afters',
  template: `
```

```
  <div class="ui label">
    <i class="list icon"></i> Counter: {{ counter }}
  </div>

  <button class="ui primary button" (click)="inc()">
    Increment
  </button>
  `
})
class AftersCmp implements OnInit, OnDestroy, DoCheck,
                           OnChanges, AfterContentInit,
                           AfterContentChecked, AfterViewInit,
                           AfterViewChecked {
  counter: number;

  constructor() {
    console.log('AfterCmd --------- [constructor]');
    this.counter = 1;
  }
  inc() {
    console.log('AfterCmd --------- [counter]');
    this.counter += 1;
  }
  ngOnInit() {
    console.log('AfterCmd - OnInit');
  }
  ngOnDestroy() {
    console.log('AfterCmp - OnDestroy');
  }
  ngDoCheck() {
    console.log('AfterCmp - DoCheck');
  }
  ngOnChanges() {
    console.log('AfterCmp - OnChanges');
  }
  ngAfterContentInit() {
    console.log('AfterCmp - AfterContentInit');
  }
  ngAfterContentChecked() {
    console.log('AfterCmp - AfterContentChecked');
  }
  ngAfterViewInit() {
    console.log('AfterCmp - AfterViewInit');
  }
  ngAfterViewChecked() {
    console.log('AfterCmp - AfterViewChecked');
  }
}
```


现在把它和Toggle按钮添加到应用组件中，就像之前在OnDestroy钩子中的用法一样。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```
  <afters *ngIf="displayAfters"></afters>
  <button class="ui primary button" (click)="toggleAfters()">
```

```
    Toggle
  </button>
```

应用组件的最终实现看起来应该是这样的。

code/advanced_components/app/ts/lifecycle-hooks/lifecycle_04.ts

```
@Component({
  selector: 'lifecycle-sample-app',
  template: `
  <h4 class="ui horizontal divider header">
    OnInit and OnDestroy
  </h4>

  <button class="ui primary button" (click)="toggle()">
    Toggle
  </button>
  <on-init *ngIf="display"></on-init>

  <h4 class="ui horizontal divider header">
    OnChange
  </h4>

  <div class="ui form">
    <div class="field">
      <label>Name</label>
      <input type="text" #namefld value="{{name}}"
             (keyup)="setValues(namefld, commentfld)">
    </div>

    <div class="field">
      <label>Comment</label>
      <textarea (keyup)="setValues(namefld, commentfld)"
                rows="2" #commentfld>{{comment}}</textarea>
    </div>
  </div>

  <on-change [name]="name" [comment]="comment"></on-change>

  <h4 class="ui horizontal divider header">
    DoCheck
  </h4>

  <do-check></do-check>

  <h4 class="ui horizontal divider header">
    AfterContentInit, AfterViewInit, AfterContentChecked and AfterViewChecked
  </h4>

  <afters *ngIf="displayAfters"></afters>
  <button class="ui primary button" (click)="toggleAfters()">
    Toggle
  </button>
  `
})
```

```
export class LifecycleSampleApp4 {
  display: boolean;
  displayAfters: boolean;
  name: string;
  comment: string;

  constructor() {
    // OnInit and OnDestroy
    this.display = true;

    // OnChange
    this.name = 'Felipe Coury';
    this.comment = 'I am learning so much!';

    // AfterXXX
    this.displayAfters = true;
  }

  setValues(namefld, commentfld) {
    this.name = namefld.value;
    this.comment = commentfld.value;
  }

  toggle(): void {
    this.display = !this.display;
  }

  toggleAfters(): void {
    this.displayAfters = !this.displayAfters;
  }
}
```

当应用启动后，我们可以看到每个钩子都打印了日志（如图14-28所示）。

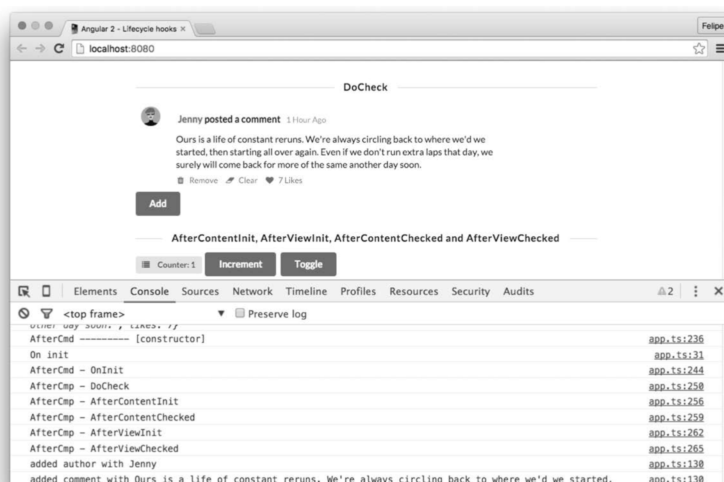


图14-28　应用启动

14

现在我们清空控制台并点击Increment按钮（如图14-29所示）。

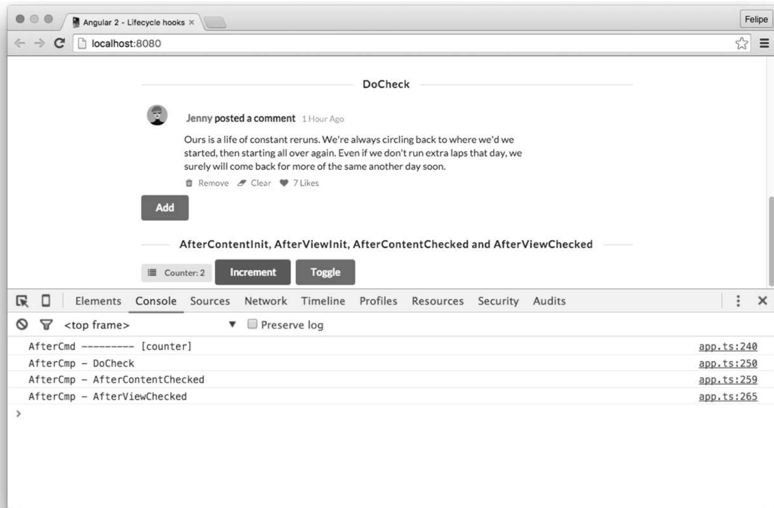


图14-29　计数增加

可以看到，这次只触发了DoCheck、AfterContentChecked和AfterViewChecked这三个钩子。

如果点击Toggle按钮，将如图14-30所示。

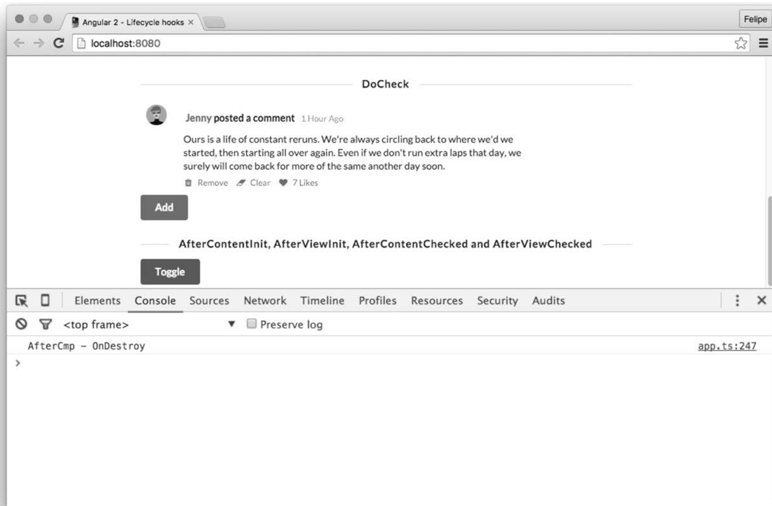


图14-30　首次切换

接着再点击一次Toggle按钮，将如图14-31所示。

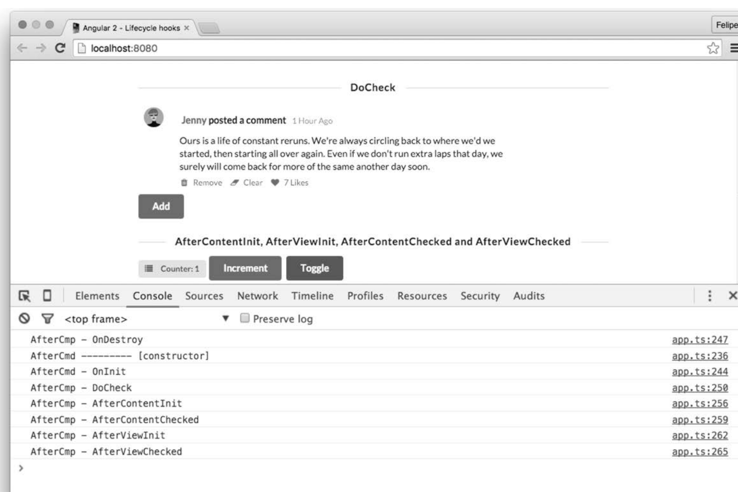


图14-31　再次切换

所有钩子都被触发了。

14.6　高级模板

template元素是种特殊的元素，用来创建可以动态操控的视图。

为了使template元素用起来更简单，Angular提供了一些语法糖来创建template元素，因此通常不需要手动创建。

举例来说，如果我们写：

```
<do-check-item
  *ngFor="let comment of comments"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</do-check-item>
```

它就会转换成：

```
<do-check-item
  template="ngFor let comment of comments; #i=index"
  [comment]="comment"
  (onRemove)="removeComment($event)">
</do-check-item>
```

接着转换成：

```
<template
  ngFor
```

```
  [ngForOf]="comments"
  let-comment="$implicit"
  let-index="i">
  <do-check-item
    [comment]="comment"
    (onRemove)="removeComment($event)">
  </do-check-item>
</template>
```

理解其背后的概念很重要，这样我们才能构建自己的指令。

14.6.1　重写 ngIf：ngBookIf

我们来创建一个指令，它和ngIf所做的事情完全一样。我们称之为ngBookIf。

1. ngBookIf的@Directive

我们先为这个类声明@Directive注解：

```
@Directive({
  selector: '[ngBookIf]',
})
```

正如前面所说，我们要使用[ngBookIf]作为选择器。这是因为当使用*ngBookIf="condition"时，它会被转换成：

```
<template ngBookIf [ngBookIf]="condition">
```

由于ngBookIf同时是一个属性，我们还需要指出想把ngBookIf作为输入属性进行接收。

这个指令要做的是：当条件为真时，添加指令模板的内容；否则删除。

当条件为真时，我们就会使用视图容器（view container）。视图容器是用来给指令附加一个或多个视图的。

视图容器可以用来：

- 创建一个新视图，嵌入我们的指令模板；
- 清空视图容器内容。

在使用它之前，需要注入ViewContainerRef和TemplateRef。它们会注入指令的视图容器和模板。

代码如下所示。

code/advanced_components/app/ts/templates/if.ts

```
class NgBookIf {
  constructor(private viewContainer: ViewContainerRef,
              private template: TemplateRef<any>) {}
```

有了视图容器和模板的引用,我们就可以写TypeScript的属性设置器(setter)了,并且用Input()注解表明它是输入属性。

code/advanced_components/app/ts/templates/if.ts

```
  @Input() set ngBookIf(condition) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.template);
    }
    else {
      this.viewContainer.clear();
    }
  }
```

每当设置类的ngBookIf属性时，这个方法都会被调用。也就是说，只要ngBookIf="condition"中的condition发生变化，就会调用这个方法。

现在，如果条件为真，就使用视图容器的createEmbeddedView方法来添加指令的模板；否则使用clear方法来删除视图容器中的所有内容。

2. 使用ngBookIf

要想使用这个指令，可以编写下面的组件。

code/advanced_components/app/ts/templates/if.ts

```
@Component({
  selector: 'template-sample-app',
  template: `
  <button class="ui primary button" (click)="toggle()">
    Toggle
  </button>

  <div *ngBookIf="display">
    The message is displayed
  </div>

  `
})
export class IfTemplateSampleApp {
  display: boolean;

  constructor() {
    this.display = true;
  }

  toggle() {
    this.display = !this.display;
  }
}
```

运行应用时，可以看到指令如预期的一样工作：当我们点击Toggle按钮时，会在页面中切换显示消息This message is displayed。

14.6.2　重写 ngFor：ngBookRepeat

现在再来编写一个简易版的ngFor指令，用来为指定的集合反复渲染模板。

1. ngBookRepeat模板解构

我们将通过*ngBookRepeat="let var of collection"语法来使用该指令。

就像在前一个指令中所做的那样，我们需要声明选择器[ngBookRepeat]。不过，这里的输入参数并不是只有ngBookRepeat。

如果回头看一下Angular是如何转换*something="let var in collection"标记的，就会发现该元素展开后的最终形态等价于：

```
<template something [somethingOf]="collection" let-var="$implicit">
  <!-- ... -->
</template>
```

如前所见，传入的输入属性不是something，而是somethingOf。它的值就是我们的指令要接收并迭代的集合。

对于生成的模板，我们将使用局部视图变量#var，它会从局部变量$implicit接收值。当Angular对语法糖进行展开时，会将一个局部变量放到模板中。这个局部变量的名称就是$implicit。

2. ngBookRepeat的@Directive

该开始编写这个指令了。首先来写指令的注解。

code/advanced_components/app/ts/templates/for.ts

```
@Directive({
  selector: '[ngBookRepeat]'
 })
```

3. ngBookRepeat类

然后编写组件类。

code/advanced_components/app/ts/templates/for.ts

```
class NgBookRepeat implements DoCheck {
  private items: any;
  private differ: IterableDiffer;
  private views: Map<any, ViewRef> = new Map<any, ViewRef>();


  constructor(private viewContainer: ViewContainerRef,
              private template: TemplateRef<any>,
              private changeDetector: ChangeDetectorRef,
              private differs: IterableDiffers) {}
```

我们为类声明了一些属性：

- items保存我们要迭代的集合；
- differ是一个IterableDiffer对象（已经在14.5节学过），用于变更检测；
- views是一个Map，它将把集合中给出的条目和包含它的视图链接起来。

构造函数会接收viewContainer、template和一个IterableDiffers实例（全部参数都在本章的前面讨论过）。

接下来要做的就是注入变更检测器。我们会在下一节中深入讲解变更检测器，现在可以先把它理解为Angular创建的类，用来在指令属性发生变化时触发检测动作。

下一步是编写设置ngBookRepeatOf属性时要触发的代码。

code/advanced_components/app/ts/templates/for.ts

```
@Input() set ngBookRepeatOf(items) {
  this.items = items;
  if (this.items && !this.differ) {
    this.differ = this.differs.find(items).create(this.changeDetector);
  }
}
```

当设置该属性时，我们将此集合保存在指令的item属性中。如果集合是有效的并且还没有differ的话，就创建一个differ。

要做到这一点，我们创建一个IterableDiffer类的实例。它可以复用指令的变更检测器（已经在构造函数中注入过了）。

接下来就要编写对集合的变化作出响应的代码了。为此，我们要实现下面的ngDoCheck方法来实现DoCheck生命周期钩子。

code/advanced_components/app/ts/templates/for.ts

```
ngDoCheck(): void {
  if (this.differ) {
    let changes = this.differ.diff(this.items);
    if (changes) {

      changes.forEachAddedItem((change) => {
        let view = this.viewContainer.createEmbeddedView(this.template,
          {'$implicit': change.item});
        this.views.set(change.item, view);
      });
      changes.forEachRemovedItem((change) => {
        let view = this.views.get(change.item);
        let idx = this.viewContainer.indexOf(view);
        this.viewContainer.remove(idx);
        this.views.delete(change.item);
      });
    }
  }
}
```

我们来分解一下这段代码。在这个方法中，我们做的第一件事就是确保differ已经实例化了。如果没有，那我们就不做任何事。

接下来，询问differ哪些东西发生了变化。如果有变化，就用`changes.forEachAddedItem`方法来遍历所有新增项。对于每个添加进来的元素，该回调方法将接收一个`CollectionChange-Record`对象。

对于每个元素，都使用视图容器的`createEmbeddedView`方法来创建一个新的嵌入视图：

```
let view = this.viewContainer.createEmbeddedView(this.template, {'$implicit': change.item});
```

`createEmbeddedView`方法的第二个参数是视图的上下文。在这个例子中，我们把局部变量`$implicit`设置为`change.item`。这样就可以访问视图里在`*ngBookRepeat="let var of collection"`中声明的`var`变量了。也就是说，`let var`中的`var`就是`$implicit`变量。使用`$implicit`是因为当我们写这个组件时还不知道用户会给它起什么名字。

最后，我们要把集合中的条目和视图关联起来。背后的原因是，如果从集合中删除了一个条目，也需要删除相应的视图。这就是接下来我们要做的。

对于从集合中删除的每一个条目，我们都要根据集合条目到视图的映射找到视图，并查询该视图在视图容器中的索引。这是因为视图容器的`remove`方法需要一个索引。最后，还要从集合条目到视图的映射中删除这个视图。

4. 试用这个指令

要测试这个指令，可以编写如下组件。

code/advanced_components/app/ts/templates/for.ts

```
@Component({
  selector: 'template-sample-app',
  template: `
  <ul>
    <li *ngBookRepeat="let p of people">
      {{ p.name }} is {{ p.age }}
      <a href (click)="remove(p)">Remove</a>
    </li>
  </ul>

  <div class="ui form">
    <div class="fields">
      <div class="field">
        <label>Name</label>
        <input type="text" #name placeholder="Name">
      </div>
      <div class="field">
        <label>Age</label>
        <input type="text" #age placeholder="Age">
      </div>
```

```
    </div>
  </div>
  <div class="ui submit button"
       (click)="add(name, age)">
    Add
  </div>
  `
})
export class ForTemplateSampleApp {
  people: any[];

  constructor() {
    this.people = [
      {name: 'Joe', age: 10},
      {name: 'Patrick', age: 21},
      {name: 'Melissa', age: 12},
      {name: 'Kate', age: 19}
    ];
  }

  remove(p) {
    let idx: number = this.people.indexOf(p);
    this.people.splice(idx, 1);
    return false;
  }

  add(name, age) {
    this.people.push({name: name.value, age: age.value});
    name.value = '';
    age.value = '';
  }
}
```

我们使用指令来遍历人员列表。

code/advanced_components/app/ts/templates/for.ts

```
<ul>
  <li *ngBookRepeat="let p of people">
    {{ p.name }} is {{ p.age }}
    <a href (click)="remove(p)">Remove</a>
  </li>
</ul>
```

当点击Remove按钮时，我们将该条目从集合中删除并触发变更检测。

我们还提供了一个表单，可以用它向集合中添加条目。

code/advanced_components/app/ts/templates/for.ts

```
<div class="ui form">
  <div class="fields">
    <div class="field">
      <label>Name</label>
      <input type="text" #name placeholder="Name">
```

```
      </div>
      <div class="field">
        <label>Age</label>
        <input type="text" #age placeholder="Age">
      </div>
    </div>
  </div>
  <div class="ui submit button"
       (click)="add(name, age)">
    Add
  </div>
```

14.7　变更检测

在用户与我们的应用交互时，数据（state）会发生改变，我们的应用需要据此作出响应。

任何现代JavaScript框架都需要解决的一大问题就是:怎样才能知道发生了变化并据此重新渲染组件?

为了让视图可以响应组件状态的变化，Angular使用了*变更检测*。

什么可以触发组件状态的改变？最明显的就是用户交互。比如，如果我们有这样一个组件：

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  <button (click)="changeName()">Change!</button>
  `
})
class MyComponent {
  name: string;
  constructor() {
    this.name = 'Felipe';
  }

  changeName() {
    this.name = 'Nate';
  }
}
```

可以看到，当用户*点击*Change!按钮时，组件的name属性会发生改变。

另一个变化的来源可能是HTTP请求：

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  `
})
class MyComponent {
```

```
  name: string;
  constructor(private http: Http) {
    this.http.get('/names/1')
      .map(res => res.json())
      .subscribe(data => this.name = data.name);
  }
}
```

最后，我们还可以用计时器来触发变化：

```
@Component({
  selector: 'my-component',
  template: `
  Name: {{name}}
  `
})
class MyComponent {
  name: string;
  constructor() {
    setTimeout(() => this.name = 'Felipe', 2000);
  }
}
```

但是Angular要如何察觉到这些变化呢？

首先要知道的是，每个组件都有自己的变更检测器。

就像我们之前看到的，一个典型的应用有很多组件，组件之间会进行交互，从而创建一个如图14-32所示的依赖关系树。

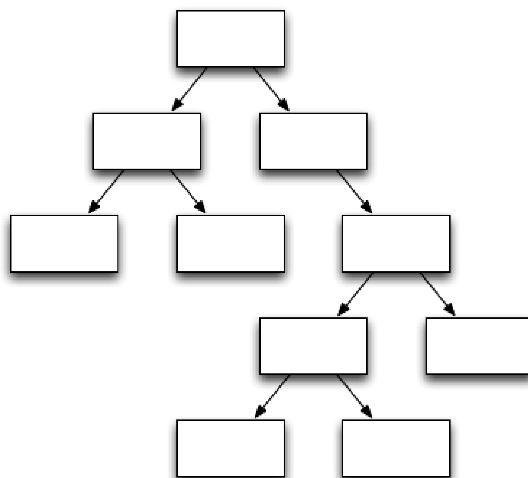


图14-32 组件树

对于树中的每个组件，都会创建一个变更检测器。因此，我们的变更检测器同样是一棵树（如图14-33所示）。

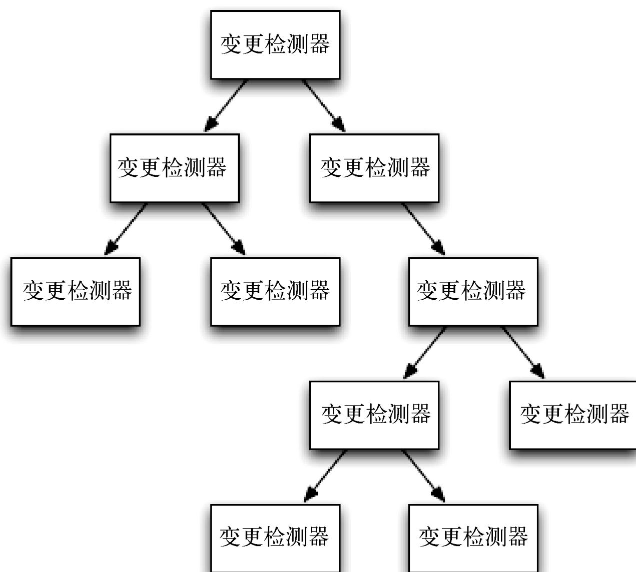


图14-33　变更检测器树

当一个组件发生变更时，无论它在树的什么位置，都会触发树中的所有变更检测器。这是因为Angular会从顶部节点开始，一直扫描到树的叶子节点（如图14-34所示）。

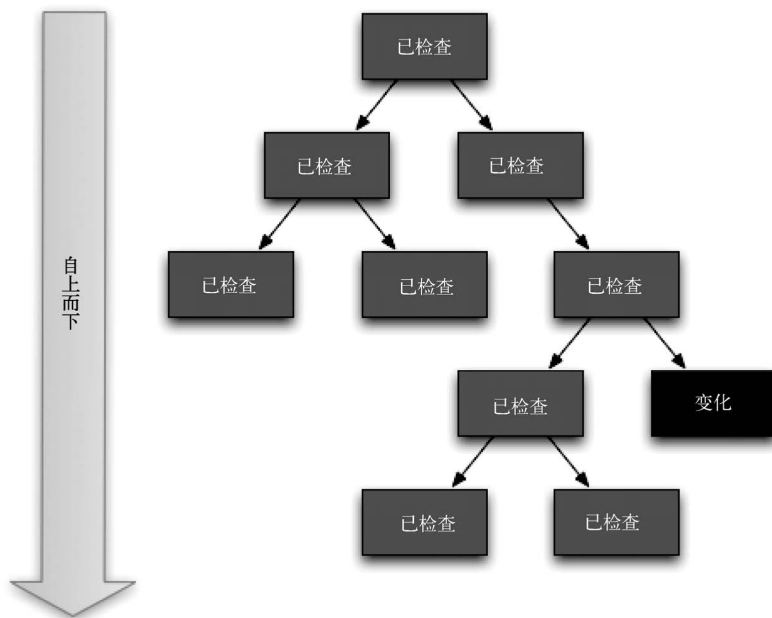


图14-34　默认的变更检测方式

在上面的图中，深灰色的组件发生了变化。但是，正如我们所见，它触发了整棵组件树中的检查。被检查的组件用浅灰色表示（注意，引起变化的组件本身也被检查了）。

直觉上，你可能会认为这种方式的开销非常大；然而实际上，由于经过大量的优化（这使得Angular代码可以被JavaScript引擎进一步优化），它的速度快得惊人。

14.7.1 自定义变更检测

有时，默认的变更检测机制可能有些大材小用。比如，你可能使用了不可改变对象或者应用的数据架构是依赖可观察对象的。在这些场景下，Angular提供了可以自定义变更检测系统的机制，可以使检测变得非常快。

修改变更检测器行为的第一种方式是告诉组件：只有当它的输入属性值发生改变时才需要去检查。

简单来说，输入属性值就是组件从外部接收的属性。比如，在这段代码中：

```
class Person {
  constructor(public name: string, public age: string) {}
}

@Component({
  selector: 'mycomp',
  template: `
    <div>
      <span class="name">{person.name}</span>
      is {person.age} years old.
    </div>
  `
})
class MyComp {
  @Input() person:Person;
}
```

我们有一个输入属性`person`。现在，如果只想在输入属性发生变化时才让组件改变，只要修改变更检测策略，把`changeDetection`设置成`ChangeDetectionStrategy.OnPush`就可以了。



顺便一提，`changeDetection`的默认值是`ChangeDetectionStrategy.Default`。

我们写两个组件来做个小实验。第一个组件使用默认的变更检测行为，而另外一个组件使用`OnPush`策略。

code/advanced_components/app/ts/change-detection/onpush.ts

```
import {
  Component,
  Input,
```

```
  ChangeDetectionStrategy,
} from '@angular/core';


class Profile {
  constructor(private first: string, private last: string) {}

  lastChanged() {
    return new Date();
  }
}
```

我们先导入一些东西，然后声明Person类。Person类会作为这两个组件的输入属性。注意，我们还在Profile类中创建了一个lastChange()方法。这个方法非常有用，可以决定何时触发变更检测。当把一个给定的组件标记为需要检查时，这个方法就会被调用，然后呈现在模板中。因此，该方法可以准确地表明组件的最后检查时间。

接下来，我们声明了DefaultCmp组件，它将使用默认变更检测策略。

code/advanced_components/app/ts/change-detection/onpush.ts

```
@Component({
  selector: 'default',
  template: `
  <h4 class="ui horizontal divider header">
    Default Strategy
  </h4>

  <form class="ui form">
    <div class="field">
      <label>First Name</label>
      <input
        type="text"
        [(ngModel)]="profile.first"
        name="first"
        placeholder="First Name">
    </div>
    <div class="field">
       <label>Last Name</label>
       <input
         type="text"
         [(ngModel)]="profile.last"
         name="last"
         placeholder="Last Name">
    </div>
  </form>
  <div>
    {{profile.lastChanged() | date:'medium'}}
  </div>
  `
})
export class DefaultCmp {
  @Input() profile: Profile;
}
```

第二个组件使用OnPush策略。

code/advanced_components/app/ts/change-detection/onpush.ts

```
@Component({
  selector: 'on-push',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <h4 class="ui horizontal divider header">
    OnPush Strategy
  </h4>

    <form class="ui form">
      <div class="field">
        <label>First Name</label>
        <input
          type="text"
          [(ngModel)]="profile.first"
          name="first"
          placeholder="First Name">
      </div>
      <div class="field">
        <label>Last Name</label>
        <input
          type="text"
          [(ngModel)]="profile.last"
          name="last"
          placeholder="Last Name">
      </div>
    </form>
    <div>
      {{profile.lastChanged() | date:'medium'}}
    </div>
  `
})
export class OnPushCmp {
  @Input() profile: Profile;
}
```

正如我们所见，两个组件使用相同的模板。唯一不同的就是注解中的变更检测策略。

最后，我们添加一个组件来并排渲染两个组件。

code/advanced_components/app/ts/change-detection/onpush.ts

```
@Component({
  selector: 'change-detection-sample-app',
  template: `
  <div class="ui page grid">
    <div class="two column row">
      <div class="column area">
        <default [profile]="profile1"></default>
      </div>
      <div class="column area">
        <on-push [profile]="profile2"></on-push>
      </div>
```

```
    </div>
  </div>
  `
})
export class OnPushChangeDetectionSampleApp {
  profile1: Profile = new Profile('Felipe', 'Coury');
  profile2: Profile = new Profile('Nate', 'Murray');
}
```

运行这个应用时，我们会看到两个组件如图14-35这样被渲染出来。

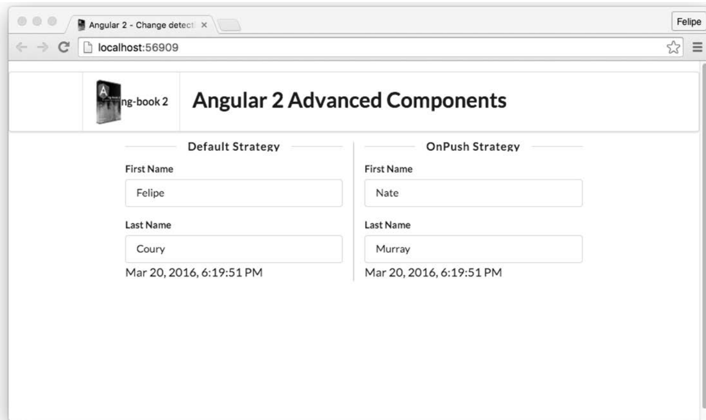


图14-35 默认策略与OnPush策略

当我们更改左边的组件（使用默认策略）时，可以注意到右边组件的时间戳并没有发生改变，如图14-36所示。

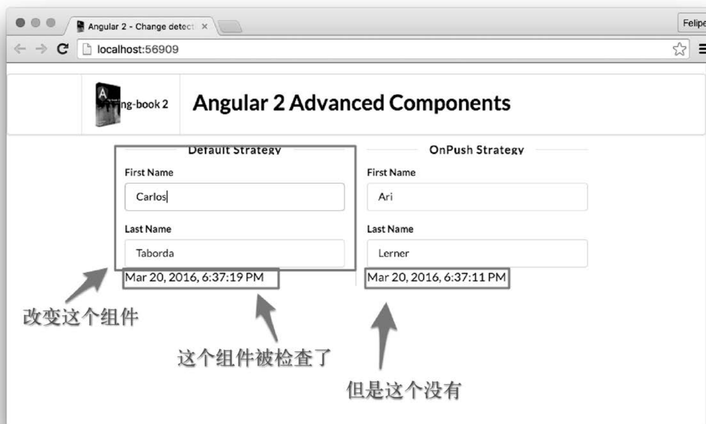


图14-36 默认的组件变化时，OnPush的组件不会检查

要理解为何如此，我们来检查下这个新的组件树（如图14-37所示）。

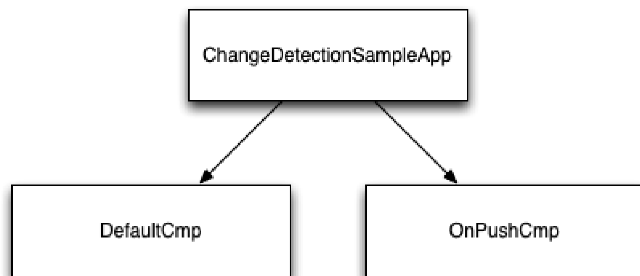


图14-37 新组件树

Angular对于变化的检查是自上而下的，所以首先查询的是ChangeDetectionSampleApp，然后是DefaultCmp，最后是OnPushCmp。当它推测出OnPushCmp发生变化时，就会自上而下地更新组件树中的所有组件，这会导致重新渲染DefaultCmp。

当我们改变右边组件的值时，如图14-38所示。



图14-38 OnPush的组件变化时，默认的组件也会检查

变更检测引擎生效后，只检查了DefaultCmp组件而没有检查OnPushCmp。这是因为当我们为组件设置了OnPush策略时，只有它自己的输入发生变化时才执行检测。改变组件树中的其他组件时并不会触发这个组件变更检测器。

14.7.2 Zones

在底层，Angular使用了一个名叫Zones的类库，它可以自动检测变化并触发变更检测机制。

在一些最常见的情景下，Zones会自动告诉Angular发生了某些变化：

- 当DOM事件发生时（比如click、change等）；
- 当HTTP请求完成时；
- 当定时器被触发时（setTimeout或setInterval）。

然而，还有一些场景是Zones无法自动识别出变化的。在这些场景下，OnPush策略就会变得非常有用。

下面是一些Zones无法掌控的例子：

- 使用异步方式运行第三方类库；
- 不可变的数据；
- 可观察对象。

在这些情况下，非常适合通过OnPush以及一点小技巧去手动提示Angular有东西发生了变化。

14.7.3　可观察对象和 OnPush

我们来编写一个组件，它接收一个可观察对象作为参数。每当从这个可观察对象中接收到值时，我们就会增加组件的计数器属性。

如果使用常规的变更检测策略，那么只要我们增加计数，Angular就会触发变更检测。然而，这个组件将使用OnPush策略，只有当计数是5的倍数或者可观察对象完成时，我们才让变更检测器生效，而不是每次增加计数时都触发变更检测器。

要做到这一点，我们来写个组件。

code/advanced_components/app/ts/change-detection/observables.ts

```
import {
  Component,
  Input,
  ChangeDetectorRef,
  ChangeDetectionStrategy
} from '@angular/core';

import { Observable } from 'rxjs/Rx';

@Component({
  selector: 'observable',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
export class ObservableCmp {
```

```
  @Input() items: Observable<number>;
  counter = 0;

  constructor(private changeDetector: ChangeDetectorRef) {
  }

  ngOnInit() {
    this.items.subscribe((v) => {
      console.log('got value', v);
      this.counter++;
      if (this.counter % 5 == 0) {
        this.changeDetector.markForCheck();
      }
    },
    null,
    () => {
      this.changeDetector.markForCheck();
    });
  }
}
```

我们将代码分解来看，以确保理解正确。首先，我们声明该组件接收items作为输入属性并使用OnPush作为变更检测策略。

code/advanced_components/app/ts/change-detection/observables.ts

```
@Component({
  selector: 'observable',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
  <div>
    <div>Total items: {{counter}}</div>
  </div>
  `
})
```

接下来，我们把输入属性存储在组件类的items属性中，然后设置另一个属性counter为0。

code/advanced_components/app/ts/change-detection/observables.ts

```
export class ObservableCmp {
  @Input() items: Observable<number>;
  counter = 0;
```

然后，我们使用构造函数来取得组件的变更检测器。

code/advanced_components/app/ts/change-detection/observables.ts

```
  constructor(private changeDetector: ChangeDetectorRef) {
  }
```

然后，当组件初始化时，在ngOnInit钩子中如下所示。

code/advanced_components/app/ts/change-detection/observables.ts

```
  ngOnInit() {
```

```
    this.items.subscribe((v) => {
      console.log('got value', v);
      this.counter++;
      if (this.counter % 5 == 0) {
        this.changeDetector.markForCheck();
      }
    },
    null,
    () => {
      this.changeDetector.markForCheck();
    });
  }
```

我们订阅了可观察对象。subscribe方法接收三个回调函数：onNext、onError和onCompleted。

onNext回调函数会打印出我们得到的值，然后增加计数。最后，如果当前计数器的值是5的倍数，我们就调用变更检测器的markForCheck方法。只要我们想告诉Angular已经发生了变化，就可以使用这个方法，从而使变更检测器生效。

对于onError回调函数，我们传入了null。这表示我们不想处理这个场景。

最后，对于onComplete回调函数，我们同样触发了变更检测器，所以最终的计数器可以被显示出来。

现在来看应用组件的代码。它会创建订阅者。

code/advanced_components/app/ts/change-detection/observables.ts

```
@Component({
  selector: 'change-detection-sample-app',
  template: `
  <observable [items]="itemObservable"></observable>
  `
})
export class ObservableChangeDetectionSampleApp {
  itemObservable: Observable<number>;

  constructor() {
    this.itemObservable = Observable.timer(100, 100).take(101);
  }
}
```

下面这行代码很重要：

```
this.itemObservable = Observable.timer(100, 100).take(101);
```

这一行创建了一个可观察对象，我们会通过items输入属性将这个可观察对象传递进组件。timer方法有两个参数：第一个是等待的毫秒数，第二个是间隔的毫秒数。因此，这个可观察对象会创建一系列的值。

因为我们不需要一直创建下去，所以使用了take函数，只获取前101个值。

当我们运行这段代码时，会发现每取到5个值才会更新一次计数器，并且生成了一个最终值101（如图14-39所示）。

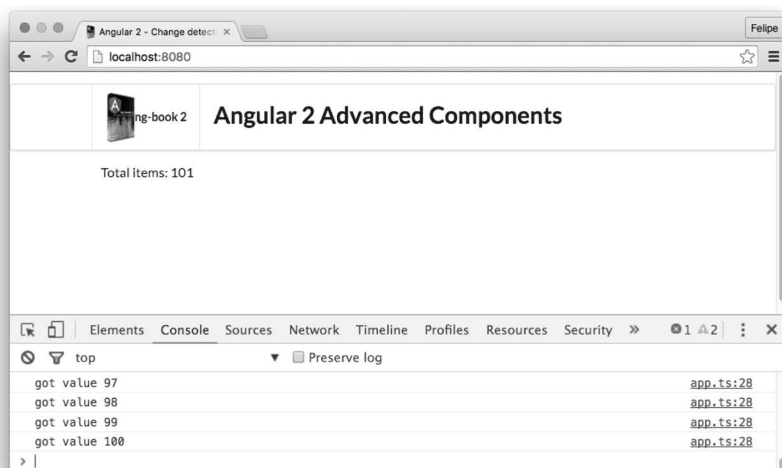


图14-39　手动触发变更检测

14.8　总结

Angular为我们提供了许多可以用来编写高级组件的工具。使用本章的这些技术，你几乎能写出任何想要的组件功能。

然而，在高级组件中还有一个重要的概念，那就是依赖注入。

使用依赖注入，我们可以让组件和系统中的很多其他部分挂接起来。第8章详细讨论了什么是依赖注入，如何在应用中使用它，以及注入服务的常用模式。

第 15 章 测试

15

经过夜以继日的奋战，终于熬到可以对外发布的日子了。是时候让过去投入的大量精力和时间得到回报了。然而，传来的一个消息犹如晴天霹雳：一个致命的bug导致用户无法注册。

15.1 测试驱动？

测试能够防患于未然，提升对程序的信心，也可以为新加入的开发人员提供指引。在软件开发领域，几乎没人质疑测试的作用。但是，人们在如何测试这个问题上一直争论不休。

一种方法是先写测试，再写实现过程，直至测试通过；另一种是已有实现代码，再写测试，验证代码是否正确。令人不解的是，二者的合理性常在开发社区中引发口水战。双方僵持不下，争论哪个才是正确的方法。

基于以往的经验，尤其是在严重依赖原型的情况下，我们将重点放在构建可测试代码上。我们发现，即使你的经历有所不同，但是在构建原型时，测试可能经常变更的代码片断会比让它运行起来耗费2~3倍的工作量。与此相反，我们在构建基于小型组件的应用程序时，将大量功能分解成不同的方法，从而测试整个蓝图的部分功能。这就是我们所说的可测试代码。



一种替代构建原型（后测试）的方法论便是所谓的“红色—绿色—重构”[1]。它的理念是要求你先写测试。运行测试会得到失败结果（红色），因为你还没有写任何实现的代码。只有在测试失败之后，才去写实现代码，直至所有测试通过（绿色）。

当然，测试什么取决于你和你的团队，而本章的重点在于讨论如何测试程序。

① Red-Green-Refactor，是一种标准的测试驱动开发流程。——译者注

15.2 端对端测试与单元测试

测试程序有两种主要方法：端对端测试和单元测试。

如果使用自上而下的方法进行测试，那么写测试时就将程序视为一个“黑盒”。与程序交互就如真实用户一样，从“旁观者”的角度评判程序是否达标。这种自上而下的测试技巧被称为端对端测试。



在Angular中，最常用的工具叫作Protractor[1]。Protractor能够打开浏览器与程序交互，收集测试结果，并检验测试结果与预期值是否相符。

第二种常用的测试方法是隔离程序的每个部件，在隔离环境中运行测试。这种测试形式叫作单元测试。

在单元测试中，所写的测试需要事先提供既定的输入值与相应的逻辑单元，检测输出结果，确定它是否与我们的预期结果匹配。

在本章中，我们将会探讨如何对Angular程序进行单元测试。

15.3 测试工具

为了测试程序，我们将用到两种工具：Jasmine和Karma。

15.3.1 Jasmine

Jasmine[2]是一种用于测试JavaScript代码的行为驱动框架。

利用Jasmine，你可以设置代码在调用后的预期结果。

比如，我们假定`Calculator`对象有一个`sum`函数。想确保1加1的结果为2，就可以用一个测试（也叫规格，spec）来表达。使用以下代码：

```
describe('Calculator', () => {
  it('sums 1 and 1 to 2', () => {
    var calc = new Calculator();
    expect(calc.sum(1, 1)).toEqual(2);
  });
});
```

使用Jasmine的一个优点是代码易于阅读。从以上代码可以看到，我们期望`calc.sum`的结果

① https://angular.github.io/protractor/#/

② http://jasmine.github.io/2.4/introduction.html

等于2。

测试通常由多个describe块和it块组成。

通常，我们用describe来组织要测试的逻辑单元，对于其内部每个要使用断言的预期都会用到一个it块。然而，这并不是一个硬性规定。你会经常看到一个it块包含多个预期。

在上述Calculator示例中，我们只是列举了一个简单的对象。正因为如此，整个类只使用了一个describe块，而每个方法使用一个it块。

大多数情况下并非如此。比如，某些方法会根据不同输入值产生不同结果，那么这些方法可以拥有多个相应的it块。在这种情况下，最好使用嵌套的describe块：对象级别用一个，每个方法也各用一个，然后在其内部的每个断言语句用一个单独的it块包裹。

大量有关describe块和it块的示例将贯穿本章。不必烦恼到底该用describe块还是it块，我们将用大量示例演示说明。

更多有关Jasmine和其语法的资料，参见Jasmine官方文档：http://jasmine.github.io/2.4/introduction.html。

15.3.2　Karma

使用Jasmine，我们可以描述测试和预期结果。要运行测试，还需要为测试提供一个浏览器环境。

Karma应运而生。使用Karma，我们可以在Chrome或Firefox之类的真实浏览器或者PhantomJS这样的空壳浏览器（无用户界面）内运行JavaScript代码。

15.4　编写单元测试

本节的重点是理解如何对一个Angular程序的各个部件进行单元测试。

我们将会学习如何测试服务、组件、HTTP请求等。同时，我们也会收获一些小技巧，让代码更容易测试。

15.5　Angular 单元测试框架

Angular自身提供了一套基于Jasmine框架的辅助类，用以帮助我们编写单元测试。

主要的测试框架位于@angular/core/testing包中。（然而，为了测试组件，我们会用到@angular/compiler/testing包和@angular/platform-browser/testing包中的一些辅助类。稍后具体介绍。）



如果这是你初次测试Angular程序，那么在为Angular写单元测试时，需要先完成一些必要的设置步骤。
例如，在需要注入依赖时，我们经常手动配置它们。在测试一个组件时，需要使用测试辅助类初始化它们。在测试路由时，还需要构建一些依赖。
设置有些繁琐，但不用太担心。一旦掌握，你就会发现从一个项目切换到另外一个项目，配置不会有多大变化。另外，本章也会指引你完成每一步。
和往常一样，可以在代码下载页面获取本章所有的源代码。用你喜欢的编辑器直接打开浏览，可以对本章涵盖的细节有一个大体的把握。我们建议你坚持参照代码来阅读本章。

15.6　测试前准备

我们在第7章创建了一个用于搜索音乐的应用。本章开始为这个程序编写测试。

Karma需要一个配置文件才能运行。因此配置Karma的第一步就是创建一个karma.conf.js文件。

将karma.conf.js放在项目的根目录下，如下所示。

code/routes/music/karma.conf.js

```
// Karma configuration
var path = require('path');
var cwd = process.cwd();

module.exports = function(config) {
  config.set({
    // base path that will be used to resolve all patterns (eg. files, exclude)
    basePath: '',

    // frameworks to use
    // available frameworks: https://npmjs.org/browse/keyword/karma-adapter
    frameworks: ['jasmine'],

    // list of files / patterns to load in the browser
    files: [
      { pattern: 'test.bundle.js', watched: false }
    ],

    // list of files to exclude
    exclude: [
    ],

    // preprocess matching files before serving them to the browser
    // available preprocessors: https://npmjs.org/browse/keyword/karma-preproces\
sor
    preprocessors: {
      'test.bundle.js': ['webpack', 'sourcemap']
```

```
    },

    webpack: {
      devtool: 'inline-source-map',
      resolve: {
        root: [path.resolve(cwd)],
        modulesDirectories: ['node_modules', 'app', 'app/ts', 'test', '.'],
        extensions: ['', '.ts', '.js', '.css'],
        alias: {
          'app': 'app'
        }
      },
      module: {
        loaders: [
          { test: /\.ts$/, loader: 'ts-loader', exclude: [/node_modules/]}
        ]
      },
      stats: {
        colors: true,
        reasons: true
      },
      watch: true,
      debug: true
    },

    webpackServer: {
      noInfo: true
    },


    // test results reporter to use
    // possible values: 'dots', 'progress'
    // available reporters: https://npmjs.org/browse/keyword/karma-reporter
    reporters: ['spec'],


    // web server port
    port: 9876,


    // enable / disable colors in the output (reporters and logs)
    colors: true,


    // level of logging
    // possible values: config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WAR\
N || config.LOG_INFO || config.LOG_DEBUG
    logLevel: config.LOG_INFO,


    // enable / disable watching file and executing tests whenever any file chan\
ges
    autoWatch: true,
```

```
    // start these browsers
    // available browser launchers: https://npmjs.org/browse/keyword/karma-launc\
her
    browsers: ['Chrome'],


    // Continuous Integration mode
    // if true, Karma captures browsers, runs the tests and exits
    singleRun: false
  })
}
```

先别急于弄清这个文件的内容，而是记住以下几点：

- 将PhantomJS设置成目标测试浏览器；
- 使用Jasmine karma框架进行测试；
- 使用一个名为test.bundle.js的webpack bundle文件包裹所有的测试和程序代码。

下一步，新建一个名为test的文件夹，用于存放测试文件：

```
mkdir test
```

15.7　测试服务类和 HTTP

服务类在Angular程序中常以普通类的形式出现。在某种意义上说，这简化了测试，因为有时可以在不需要Angular的情况下直接进行测试。

配置好Karma，就可以开始测试SpotifyService类了。如果记得没错，这个服务类通过与Spotify API交互读取专辑、曲目和艺术家相关信息。

切换到test文件夹，新建一个service子文件夹，用于存放即将开始的服务类测试。一切准备就绪，开始创建第一个服务类测试文件，名为SpotifyService.spec.ts。

下面开始组织这个测试文件。首先需要从@angular/core/testing包中导入几个辅助类。

code/routes/music/test/services/SpotifyService.spec.ts

```
import {
  inject,
  fakeAsync,
  tick,
  TestBed
} from '@angular/core/testing';
```

接下来，还需要导入其他几个类。

code/routes/music/test/services/SpotifyService.spec.ts

```
import {MockBackend} from '@angular/http/testing';
```

```
import {
  Http,
  ConnectionBackend,
  BaseRequestOptions,
  Response,
  ResponseOptions
} from '@angular/http';
```

既然我们的服务用到了HTTP请求，就需要从@angular/http/testing包中导入MockBackend。有了这个类，就可以设置预期值和验证HTTP请求结果了。

最后，导入我们要测试目标类。

code/routes/music/test/services/SpotifyService.spec.ts

```
import {SpotifyService} from '../../app/ts/services/SpotifyService';
```

15.7.1　HTTP 要点

马上要编写测试了，但是在每个测试的运行过程中都要访问Spotify服务器。这显然有些不妥，原因如下。

(1) HTTP请求相对比较慢，而且随着测试套件的体积越来越大，可以预见运行全部测试需要的时间也会越来越长。

(2) Spotify的API调用设置了阈值限制，如果不停地运行测试，会很快耗尽所有的API调用资源。

(3) 如果处于离线状态、Spotify崩溃或无法访问，那么测试也会失败，即使代码在技术角度上没有问题也是一样。

这在写单元测试时给了我们一个提示：在运行测试前，必须隔离那些无法掌控的东西。

在我们例子中，对应的就是Spotify服务。解决方法是，用一个替身替换掉HTTP请求，而且这个替身不需要访问真实的Spotify服务器。

在测试领域，这个过程被称为模拟依赖，也时也叫作伪装依赖。



阅读文章“模拟不是伪装”（http://martinfowler.com/articles/mocksArentStubs.html）可以了解更多有关模拟和伪装之间的差异。

假设我们正在写的测试依赖于某个Car类。

它有几个方法：你可以调用start来启动一个Car实例，也可以调用汽车的其他方法，如stop（停车），park（泊车）和getSpeed（读取车速）。

下面介绍如何使用伪装和模拟来写依赖于这个类的测试。

15.7.2　伪装

伪装是即时创建的对象，它包含所依赖对象所有行为的一个子集。

下面写一个测试与start方法交互。

为Car即时创建一个伪装并将它注入到要测试的类中：

```
describe('Speedtrap', function() {
  it('tickets a car at more than 60mph', function() {
    var stubCar = { getSpeed: function() { return 61; } };
    var speedTrap = new SpeedTrap(stubCar);
    speedTrap.ticketCount = 0;
    speedTrap.checkSpeed();
    expect(speedTrap.ticketCount).toEqual(1);
  });
});
```

这是使用伪装的一个典型场景。我们可能仅仅在某个测试内部使用它。

15.7.3　模拟

在我们的例子中，模拟是对象更完整的体现，它会重写依赖的部分或全部行为。在大部分情况下， 模拟可以在一个测试套件的多个测试间反复使用。

它们有时用于断言方法是否按预期的方式调用。

一个模拟版本的Car类可能是这样的：

```
class MockCar {
  startCallCount: number = 0;

  start() {
    this.startCallCount++;
  }
}
```

它可以用在另外一个测试中，如：

```
describe('CarRemote', function() {
  it('starts the car when the start key is held', function() {
    var car = new MockCar();
    var remote = new CarRemote();
    remote.holdButton('start');
    expect(car.startCallCount).toEqual(1);
  });
});
```

模拟和伪装的最大区别在于：

- 伪装提供手动重写行为功能的一个子集；
- 模拟通常预设期望值，验证调用某些方法的返回结果。

15.7.4　Http MockBackend

既然现在心里有底了，就继续编写之前的服务类测试代码。

每次运行测试时都与在线的Spotify服务进行交互显然不是个好主意。幸运的是Angular提供了一种方法，使用MockBackend来伪装HTTP调用。

可以将这个类注入到一个Http实例中，这样我们就能按照自己的意图对HTTP交互行为进行操控了。可以使用不同的方法进行干预和断言：手动设置响应，模拟HTTP错误，添加更多预期（比如判断请求的URL是否与预期值匹配，请求参数是否正确，等等）。

因此这里的想法就是使用一个伪HTTP库。这个伪HTTP看起来和真实的HTTP库一样：所有方法一一匹配，可以返回响应结果，等等。然而，我们却不会*真正*发出一条请求。

事实上，除了伪造请求外，MockBackend还允许我们设置*期望结果*，监控我们的预期行为。

15.7.5　TestBed.configureTestingModule 和提供者

当测试Angular程序时，需要确保配置顶级NgModule，后面会在这个测试中用到它。在进行配置时，我们要配置提供者、声明组件并导入其他模块：就像你平常使用NgModule一样（参见8.10节）。

测试Angular代码时，我们有时采取*手动设置注入*的方式。这样做的好处是能够对测试进行更多的操控。

所以在测试Http请求时，我们不会注入一个"真实"的Http类，取而代之的是注入一个看起来像Http的替身，但它可以真实地拦截请求，返回我们事先配置的响应。

为了做到这一点，要创建一个Http变体，其内部使用MockBackend。

方法是，在beforeEach钩子中使用TestBed.configureTestingModule。这个钩子接收一个回调函数，它会在每个测试运行之前被调用。这为替换类的具体实现提供了一个难得的机会。

code/routes/music/test/services/SpotifyService.spec.ts

```
describe('SpotifyService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        BaseRequestOptions,
        MockBackend,
        SpotifyService,
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] },
```

```
      ]
    });
  });
```

注意TestBed.configureTestingModule的providers参数可以接收提供者数组，用于测试注入器。

BaseRequestOptions和SpotifyService是那些类的默认实现。最后一个提供者有点复杂。

code/routes/music/test/services/SpotifyService.spec.ts

```
        { provide: Http,
          useFactory: (backend: ConnectionBackend,
                       defaultOptions: BaseRequestOptions) => {
                         return new Http(backend, defaultOptions);
                       }, deps: [MockBackend, BaseRequestOptions] },
      ]
```

这段代码使用了provide和useFactory参数来创建一个Http类变体，使用了工厂模式（也就是useFactory的职责所在）。

这个工厂的方法签名需要接收一个ConnectionBackend实例和一个BaseRequestOption实例。这个对象的第二个参数是deps: [MockBackend, BaseRequestOptions]。这表示MockBackend是工厂的第一个参数，BaseRequestOptions（默认实现）为第二个参数。

最后，返回一个MockBackend作为函数结果的定制Http类。

这样做有什么好处呢？在测试代码中每次需要注入Http的地方，得到的都是我们改装过的Http实例。

我们会在大量测试中使用这个行之有效的方法：用依赖注入的方法定制依赖，隔离需要测试的功能。

15.7.6 测试 getTrack 方法

下面针对这个服务类写一个测试，验证我们正在调用正确的URL。



如果你还没看过第7章的音乐程序，可以在7.10.5节找到源代码。

现在开始测试getTrack方法。

code/routes/music/app/ts/services/SpotifyService.ts

```
  getTrack(id: string): Observable<any[]> {
    return this.query(`/tracks/${id}`);
  }
```

你可能还记得这个方法的细节，它调用了query方法，从而分析接收的参数并拼接成URL。

code/routes/music/app/ts/services/SpotifyService.ts

```
query(URL: string, params?: Array<string>): Observable<any[]> {
  let queryURL: string = `${SpotifyService.BASE_URL}${URL}`;
  if (params) {
    queryURL = `${queryURL}?${params.join('&')}`;
  }

  return this.http.request(queryURL).map((res: any) => res.json());
}
```

请求/tracks/${id}意味着假设当调用getTrack('TRACK_ID')方法时，期望返回的URL是https://api.spotify.com/v1/tracks/TRACK_ID。

可以这样写这个测试：

```
describe('getTrack', () => {
  it('retrieves using the track ID',
    inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend) => {
      var res;
      mockBackend.connections.subscribe(c => {
        expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
        let response = new ResponseOptions({body: '{"name": "felipe"}'});
        c.mockRespond(new Response(response));
      });
      spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});
```

初看有点难以理解，下面一一讲解。

当测试有依赖时，使用Angular注入器提供那些类的实例。如下所示：

```
inject([Class1, ..., ClassN], (instance1, ..., instanceN) => {
  ... testing code ...
})
```

当测试返回的是一个承诺或者RxJS的可观察对象时，可以使用fakeAsync辅助工具来测试那些代码（像测试同步代码那样）。在调用tick()后，承诺立即生效，可观察对象也会马上接收到通知。

如下列代码所示：

```
inject([SpotifyService, MockBackend], fakeAsync((spotifyService, mockBackend) => {
  ...
}));
```

首先要读取两个变量：spotifyService和mockBackend。前者是一个特定的SpotifyService实例，后者是一个MockBackend实例。注意内部函数（spotifyService, mockBackend）的参数是注入的，相应的类型在inject函数第一个参数的数组中（SpotifyService和MockBackend）指定。

其次运行位于fakeAsync内部的代码。这就意味着当调用tick()时，异步代码会以同步方式运行。

测试的运行环境已经准备就绪，现在可以写“真正”的测试代码了。首先声明一个res变量，存放HTTP调用响应结果。然后，订阅mockBackend.connections事件：

```
var res;
mockBackend.connections.subscribe(c => { ... });
```

简单地说，每当mockBackend上产生一个新的连接，我们都希望收到通知（例如，调用了这个函数）。

为了验证SpotifyService根据指定的TRACK_ID调用了正确的URL，可以指定预期结果为我们预设的URL。首先通过c.request.url得到URL值，然后设置期望结果：c.request.url的值应该是字符串'https://api.spotify.com/v1/tracks/TRACK_ID'：

```
expect(c.request.url).toBe('https://api.spotify.com/v1/tracks/TRACK_ID');
```

运行测试。如果请求URL不匹配，则测试失败。

现在我们已经收到了请求，并证明了它是正确的。现在需要打造一个响应。为此，新建一个ResponseOptions实例，指定JSON字符串{"name": "felipe"}为响应内容。

```
let response = new ResponseOptions({body: '{"name": "felipe"}'});
```

最后，将连接的响应替换成一个Response对象，它包裹了刚刚创建的ResponseOptions实例。

```
c.mockRespond(new Response(response));
```



注意，subscribe中的回调函数可以复杂到任何你想要的程度，可以包含基于URL的条件逻辑、查询参数或者任何可以从请求对象中读取的信息。

这样一来，我们就可以为可能遇到的每个场景编写测试了。

现在已经准备好了使用TRACK_ID参数来调用getTrack方法，并且可以通过res变量跟踪响应结果：

```
spotifyService.getTrack('TRACK_ID').subscribe((_res) => {
  res = _res;
});
```

如果此时中断测试，在触发回调函数前会一直等待HTTP请求发送和响应结果返回。也有可能产生其他执行路径，我们不得不重新组织代码对任务进行同步。幸好fakeAsync可以解决这个

问题。方法是调用tick()，异步代码会立即执行，就像变魔术一样：

```
tick();
```

执行最后一步检验，确保设置的响应结果和接收到的相同：

```
expect(res.name).toBe('felipe');
```

细想一下，这个服务类的所有方法的代码都非常类似。将设置URL预期值的代码片断抽取出来，放到一个名为expectedURL的函数中。

code/routes/music/test/services/SpotifyService.spec.ts

```
function expectURL(backend: MockBackend, url: string) {
  backend.connections.subscribe(c => {
    expect(c.request.url).toBe(url);
    let response = new ResponseOptions({body: '{"name": "felipe"}'});
    c.mockRespond(new Response(response));
  });
}
```

依葫芦画瓢，可以轻而易举地为getArtist和getAlbum方法编写测试。

code/routes/music/test/services/SpotifyService.spec.ts

```
describe('getArtist', () => {
  it('retrieves using the artist ID',
    inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
      var res;
      expectURL(backend, 'https://api.spotify.com/v1/artists/ARTIST_ID');
      svc.getArtist('ARTIST_ID').subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});

describe('getAlbum', () => {
  it('retrieves using the album ID',
    inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
      var res;
      expectURL(backend, 'https://api.spotify.com/v1/albums/ALBUM_ID');
      svc.getAlbum('ALBUM_ID').subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});
```

searchTrack方法稍有不同：它不直接调用query，而是使用search方法替代。

code/routes/music/app/ts/services/SpotifyService.ts

```
searchTrack(query: string): Observable<any[]> {
  return this.search(query, 'track');
}
```

search接着调用query，将/search作为第一个参数并将一个包含q=<query>和type=track的数组作为第二个参数。

code/routes/music/app/ts/services/SpotifyService.ts

```
search(query: string, type: string): Observable<any[]> {
  return this.query(`/search`, [
    `q=${query}`,
    `type=${type}`
  ]);
}
```

最后，query将参数转换成带有QueryString的URL路径。我们期待调用的URL是以/search?q=&type=track结尾的。

综合所学知识，为searchTrack方法编写测试。

code/routes/music/test/services/SpotifyService.spec.ts

```
describe('searchTrack', () => {
  it('searches type and term',
    inject([SpotifyService, MockBackend], fakeAsync((svc, backend) => {
      var res;
      expectURL(backend, 'https://api.spotify.com/v1/search?q=TERM&type=track'\
);
      svc.searchTrack("TERM").subscribe((_res) => {
        res = _res;
      });
      tick();
      expect(res.name).toBe('felipe');
    }))
  );
});
```

这个测试与之前写过的测试异曲同工。下面一起回顾这个测试的内容：

- 植入HTTP生命周期，在HTTP连接初始化时添加回调；
- 为当前连接设置预期URL，包含查询类型和搜索关键字；
- 调用测试方法searchTrack；
- 通知Angular完成所有等待的异步调用；
- 断言预期响应结果。

简而言之，测试服务类时要做的是：

(1) 使用伪装或模拟来隔离全部依赖；

(2) 在异步调用的情况下，使用fakeAsync和tick确保它们的完成；

(3) 调用要测试的服务类；

(4) 断言方法返回值与预期值匹配。

下面把注意力转向那些消费服务的类：组件。

15.8　测试组件间的路由

测试组件时，可以使用下列两种策略之一：

(1) 编写测试从外部与组件进行交互，传递属性值，检验标签生成结果；

(2) 测试各个组件方法及其输出结果。

这两种测试策略分别称为*黑盒测试*和*白盒测试*。本节将演示如何混合使用它们。

首先为相对简单的一个组件ArtistComponent类编写测试。第一部分测试将测试组件的内部结构，所以它属于*白盒测试*。

在开始之前，先回顾一下ArtistComponent的内容：

在类的构造函数上，首先从routeParams集合中读取id。

code/routes/music/app/ts/components/ArtistComponent.ts

```
  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }
```

很快,我们遇到了第一个绊脚石:在没有运行状态路由器的情况下,如何获取当前路由的ID?

15.8.1　为测试创建路由器

在Angular中写测试时，我们手动配置了大量注入的类。路由（和测试组件）也包含大量需要注入的依赖。尽管如此，一旦配置好了，它就很少变更而且简单易用。

写测试时，使用beforeEach和TestBed.configureTestingModule设置可注入的依赖是很方便的。在测试ArtistComponent的时候，将定义一个函数来创建和配置这个测试的路由。

code/routes/music/test/components/ArtistComponent.spec.ts

```
describe('ArtistComponent', () => {
  beforeEach(() => {
    configureMusicTests();
  });
```

在辅助类文件MusicTestHelpers.ts中定义方法configureMusicTests。一起来看看。

这仅仅是configureMusicTests的实现代码。不用担心，下面将逐一解释。

code/routes/music/test/MusicTestHelpers.ts

```
export function configureMusicTests() {
  const mockSpotifyService: MockSpotifyService = new MockSpotifyService();

  TestBed.configureTestingModule({
    imports: [
      { // TODO RouterTestingModule.withRoutes coming soon
        ngModule: RouterTestingModule,
        providers: [provideRoutes(routerConfig)]
      },
      TestModule
    ],
    providers: [
      mockSpotifyService.getProviders(),
      {
        provide: ActivatedRoute,
        useFactory: (r: Router) => r.routerState.root, deps: [ Router ]
      }
    ]
  });
}
```

首先创建一个MockSpotifyService实例，用来模拟真实的SpotifyService实现。

接下来，使用一个名为TestBed的类，并调用其方法configureTestingModule。TestBed是Angular内置的一个辅助类库，帮助我们简化测试。

本例中，TestBed.configureTestingModule的作用是为测试配置NgModule。你可以看到我们提供了一个NgModule配置作为参数，它包含：

- imports
- providers

在imports中，导入：

- RouterTestingModule，并用routerConfig进行配置——这样能够为测试配置路由器；
- TestModule，这个NgModule声明了所有将要测试的组件（具体细节参见MusicTestHelpers.ts）。

在providers中，提供了：

- MockSpotifyService（通过mockSpotifyService.getProviders()）
- ActivatedRoute

我们以Router为入口，进一步学习。

1. Router

至今尚未提及的是测试时要用到哪些路由。对此有多种方法实现，首先看一下我们要用的方式。

code/routes/music/test/MusicTestHelpers.ts

```
@Component({
  selector: 'blank-cmp',
  template: ``
})
export class BlankCmp {
}

@Component({
  selector: 'root-cmp',
  template: `<router-outlet></router-outlet>`
})
export class RootCmp {
}

export const routerConfig: Routes = [
  { path: '', component: BlankCmp },
  { path: 'search', component: SearchComponent },
  { path: 'artists/:id', component: ArtistComponent },
  { path: 'tracks/:id', component: TrackComponent },
  { path: 'albums/:id', component: AlbumComponent }
];
```

这里并不（像真实路由器配置的那样）跳转到一个空的URL，而是使用一个BlankCmp替代。

当然，如果你坚持像顶层应用那样使用RouterConfig，那么要先在其他地方使用export导出，并在此处使用import导入。

如果遇到更复杂的场景，必须针对多种不同的路由配置进行测试，那么可以在musicTestProviders函数中接收一个参数，从而每次运行测试都使用一个新的路由器配置。

面临太多的选择，你必须挑选一种最适合自己团队的方式。在路由是相对静态的并且一个配置可以服务于所有测试的情况下，这个配置相当棒。

现在所有依赖都已经解决，可以通过new Router创建一个新的路由器，并调用其r.initialNavigation()方法。

2. ActivatedRoute

ActivatedRoute服务跟踪“当前路由”。它需要把Router作为依赖，并把它加入到deps来进行注入。

3. MockSpotifyService

之前通过模拟HTTP库测试了SpotifyService。这里我们将会模拟整个服务类。一起来看看如何模拟这个类，或者说任何服务。

15.8.2 模拟依赖

在music/test目录下，找到mocks/spotify.ts文件，内容如下。

code/routes/music/test/mocks/spotify.ts

```
import {SpyObject} from './helper';
import {SpotifyService} from '../../app/ts/services/SpotifyService';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  fakeResponse;
```

这里声明MockSpotifyService服务类，它是真实SpotifyService的一个模拟版本。这些实例变量会被作为探子（spy）使用。

15.8.3 探子

探子是一种比较特别的模拟对象，有两个好处：

(1) 可以模拟返回值；

(2) 可以计算方法调用次数和调用的参数值。

要在Angular测试中使用探子，可以使用一个内部类SpyObject实现（用于Angular内部测试）。

正如我们的代码所示，你可以即时创建一个新SpyObject或者让模拟类继承SpyObject。

继承或直接使用这个类的好处在于，它提供一个spy方法。spy方法允许你覆盖某个方法并强制返回值（以及监控，确保方法被调用）。下面的代码对类构造函数使用spy。

code/routes/music/test/mocks/spotify.ts

```
  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').andReturn(this);
    this.getArtistSpy = this.spy('getArtist').andReturn(this);
    this.getTrackSpy = this.spy('getTrack').andReturn(this);
    this.searchTrackSpy = this.spy('searchTrack').andReturn(this);
  }
```


构造函数的第一行调用了SpyObject构造函数，传递要模拟的特定类。调用super(...)是可选的，但模拟时类会继承所有特定类的方法，因此你只需要覆盖要测试的方法。



如果你想知道SpyObject是如何实现的，请查看angular/angular项目下的文件/modules/angular2/src/testing/testing_internal.ts（https://github.com/angular/angular/blob/b0cebd-ba6b651e9e7eb5bf801ea42dc7c4a7f25/modules/angular2/src/testing/testing_internal.ts #L205）。

调用super之后，将fakeResponse的值初始化为null，我们稍后会用到它。

接下来用探子替换特定类的方法。编写测试时使用一个引用更容易设置预期值和模拟响应结果。

在ArtistComponent中使用SpotifyService时，真实的getArtist方法返回一个可观察对象。在组件中调用的方法是subscribe方法。

code/routes/music/app/ts/components/ArtistComponent.ts

```
ngOnInit(): void {
  this.spotify
    .getArtist(this.id)
    .subscribe((res: any) => this.renderArtist(res));
}
```

然而在模拟类中，我们会采取一个小技巧：getArtist并不返回可观察对象，而是返回this，也就是MockSpotifyService自身。这就意味着上面this.spotify.getArtist(this.id)的返回值是MockSpotifyService。

不过这样有一个问题：ArtistComponent将会调用可观察对象的subscribe方法。考虑到这一点，可以在MockSpotifyService中定义一个subscribe方法。

code/routes/music/test/mocks/spotify.ts

```
subscribe(callback) {
  callback(this.fakeResponse);
}
```

现在在模拟对象上调用subscribe方法，会立即调用这个回调函数，异步方法会同步执行。

另外注意，我们使用this.fakeResponse来调用这个回调函数。它将我们引向另一个方法。

code/routes/music/test/mocks/spotify.ts

```
setResponse(json: any): void {
  this.fakeResponse = json;
}
```

这个方法并没有替换特定服务的任何部件，取而代之的是使用一个辅助方法，允许测试代码设置既定的响应结果（可能来源于特定的服务），并利用它模拟不同的响应。

code/routes/music/test/mocks/spotify.ts

```
getProviders(): Array<any> {
  return [{ provide: SpotifyService, useValue: this }];
}
```

最后一个方法是辅助方法，用在TestBed.configureTestingModule的providers参数上。它和稍后回过头来写组件测试时看到的类似。

下面是MockSpotifyService的完整代码。

code/routes/music/test/mocks/spotify.ts

```
import {SpyObject} from './helper';
import {SpotifyService} from '../../app/ts/services/SpotifyService';

export class MockSpotifyService extends SpyObject {
  getAlbumSpy;
  getArtistSpy;
  getTrackSpy;
  searchTrackSpy;
  mockObservable;
  fakeResponse;

  constructor() {
    super(SpotifyService);

    this.fakeResponse = null;
    this.getAlbumSpy = this.spy('getAlbum').andReturn(this);
    this.getArtistSpy = this.spy('getArtist').andReturn(this);
    this.getTrackSpy = this.spy('getTrack').andReturn(this);
    this.searchTrackSpy = this.spy('searchTrack').andReturn(this);
  }

  subscribe(callback) {
    callback(this.fakeResponse);
  }

  setResponse(json: any): void {
    this.fakeResponse = json;
  }

  getProviders(): Array<any> {
    return [{ provide: SpotifyService, useValue: this }];
  }
}
```

15.9 回到测试代码

万事俱备，只欠东风。现在可以为ArtistComponent编写测试代码了。

首先是导入语句。

15

code/routes/music/test/components/ArtistComponent.spec.ts

```
import {
  inject,
  fakeAsync,
```

```
} from '@angular/core/testing';
import { Router } from '@angular/router';
import { Location } from '@angular/common';
import { MockSpotifyService } from '../mocks/spotify';
import { SpotifyService } from '../../app/ts/services/SpotifyService';
import {
  advance,
  createRoot,
  RootCmp,
  configureMusicTests
} from '../MusicTestHelpers';
```

接下来使用configureMusicTests描述测试，确保所有测试用例都可以访问musicTestProviders。

code/routes/music/test/components/ArtistComponent.spec.ts

```
describe('ArtistComponent', () => {
  beforeEach(() => {
    configureMusicTests();
  });
```

然后写一个测试来验证组件初始化的细节。首先，回顾一下ArtistComponent的初始化过程。

code/routes/music/app/ts/components/ArtistComponent.ts

```
export class ArtistComponent implements OnInit {
  id: string;
  artist: Object;

  constructor(private route: ActivatedRoute, private spotify: SpotifyService,
              private location: Location) {
    route.params.subscribe(params => { this.id = params['id']; });
  }

  ngOnInit(): void {
    this.spotify
      .getArtist(this.id)
      .subscribe((res: any) => this.renderArtist(res));
  }
```

请记住，创建组件时，使用route.params接收当时路由的id参数，并将它存储在类的id属性中。

当组件初始化时，ngOnInit方法被Angular触发（因为此组件实现了OnInit接口）。然后针对接收到的id使用SpotifyService读取相应的艺术家。当获取艺术家数据后，调用renderArtist，传递艺术家数据。

这里一个重要的理念就是使用依赖注入来获取SpotifyService，但是要记得，我们之前已经创建了一个MockSpotifyService。

为了测试这一行为，执行以下步骤：

(1) 使用路由导向到ArtistComponent，组件会进行初始化；

(2) 验证MockSpotifyService在ArtistComponent中已经被注入，根据相应的id读取艺术家数据。

下面是完整的测试代码。

code/routes/music/test/components/ArtistComponent.spec.ts

```
describe('initialization', () => {
  it('retrieves the artist', fakeAsync(
    inject([Router, SpotifyService],
           (router: Router,
            mockSpotifyService: MockSpotifyService) => {
      const fixture = createRoot(router, RootCmp);

      router.navigateByUrl('/artists/2');
      advance(fixture);

      expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
    })));
});
```

接下来一步步进行解释。

15.9.1 fakeAsync 和 advance

首先用fakeAsync包裹测试。有了fakeAsync，我们就能够在状态检测和异步操作发生时进行更多的控制，并且不需要深入其内部细节。这样做的结果是，我们在测试中做了变更，必须显式地通知组件检测变更结果。

一般来说，开发程序时不必担心这个问题，这是Zones应该做的事。但在整个测试过程中，我们可以更细致地对状态变化的检测进行操作。

向下跳过几行，可以看到调用了MusicTestHelpers中的advance函数。一起看看这个函数。

code/routes/music/test/MusicTestHelpers.ts

```
export function advance(fixture: ComponentFixture<any>): void {
  tick();
  fixture.detectChanges();
}
```

advance函数做了两件事：

(1) 通知组件检测状态变更；

(2) 调用tick()。

使用fakeAsync时，计时器是同步的。我们使用tick()来模拟异步流逝的时间。

实际上，在我们的测试中，任何需要Angular大显身手的时候都可以调用advance函数。例如，如果要导向到新的路由，更新一个表单元素，发出一个HTTP请求等，我们都可以调用advance函数给Angular制造机会大显神通。

15.9.2　inject

在测试中需要添加一些依赖。使用inject可以做到这一点。inject接收两个参数：

(1) 一个等待注入的令牌数组

(2) 一个提供了注入的函数

inject会使用哪些类？提供者通过TestBed.configureTestingModule的providers来定义。

注意，这里要注入：

(1) Router

(2) SpotifyService

要注入的Router类就是上面musicTestProviders中配置的Router。

对于SpotifyService，注意请求注入SpotifyService时，得到的是MockSpotifyService。看起来有点晦涩，但是基于目前为止的讨论你应该可以理解。

15.9.3　测试 ArtistComponent 组件初始化

一起回顾一下测试代码的内容。

code/routes/music/test/components/ArtistComponent.spec.ts

```
      const fixture = createRoot(router, RootCmp);

      router.navigateByUrl('/artists/2');
      advance(fixture);

      expect(mockSpotifyService.getArtistSpy).toHaveBeenCalledWith('2');
```

我们使用createRoot创建一个RootCmp实例。一起看看createRoot辅助函数。

code/routes/music/test/MusicTestHelpers.ts

```
export function createRoot(router: Router,
                           componentType: any): ComponentFixture<any> {
  const f = TestBed.createComponent(componentType);
  advance(f);
  (<any>router).initialNavigation();
  advance(f);
  return f;
}
```

注意，这时调用createRoot可以：

(1) 创建一个根组件实例；

(2) 对其进行advance处理；

(3) 通知路由器设置initialNavigation；

(4) 再次进行advance处理；

(5) 返回全新的根组件。

测试依赖路由器的组件时需要不少准备工作，有这个辅助函数可以方便不少。

注意我们再次使用了TestBed类库来调用TestBed.createComponent方法。这个方法创建了一个相应类型的组件。



RootCmp是我们在MusicTestHelpers中创建的一个空组件。其实完全没必要为根组件创建一个空组件。这里这样做是因为能够或多或少在隔离环境中测试子组件（ArtistComponent）。这样一来，就不必担心对上层应用组件的影响了。但是，也许你**想要**确保子组件在上下文环境中正确运行。在这种情况下，你可能想使用应用程序常规的父组件，而非RootCmp。

接下来探讨使用router转向URL /artists/2以及advance。当我们定位到该URL时，ArtistComponent应该会进行初始化，因此可以断定调用SpotifyService的getArtist方法时返回了正确的值。

15.9.4 测试 ArtistComponent 方法

回想一下，ArtistComponent中有一个href调用了back()方法。

code/routes/music/app/ts/components/ArtistComponent.ts

```
back(): void {
  this.location.back();
}
```

下面来测试，当调用back方法时，路由器会将用户重定向回之前的位置。

当前位置状态由Location服务控制。当需要将用户返回原来的位置时，我们使用Location的back方法。

这里演示如何测试back方法。

code/routes/music/test/components/ArtistComponent.spec.ts

```
describe('back', () => {
  it('returns to the previous location', fakeAsync(
    inject([Router, Location],
           (router: Router, location: Location) => {
      const fixture = createRoot(router, RootCmp);
      expect(location.path()).toEqual('/');

      router.navigateByUrl('/artists/2');
```

```
      advance(fixture);
      expect(location.path()).toEqual('/artists/2');

      const artist = fixture.debugElement.children[1].componentInstance;
      artist.back();
      advance(fixture);

      expect(location.path()).toEqual('/');
    })));
});
```

初始结构与之类似：注入依赖并创建一个新的组件。

加入一个新的expect语句，断定location.path()与预期结果一致。

这里提供一种新思路：当访问ArtistComponent的方法时，通过fixture.debugElement.children[1].componentInstance这一行得到ArtistComponent实例的一个引用。

有了组件实例，就可以直接调用其方法了，如back()。

调用了back方法后，进行advance处理，然后验证location.path()是否与预期一致。

15.9.5　测试 ArtistComponent DOM 模板值

最后要测试ArtistComponent的一部分就是生成艺术家模板。

code/routes/music/app/ts/components/ArtistComponent.ts

```
template: `
<div *ngIf="artist">
  <h1>{{ artist.name }}</h1>

  <p>
    <img src="{{ artist.images[0].url }}">
  </p>

  <p><a href (click)="back()">Back</a></p>
</div>
`
```

记住，实例变量artist是SpotifyService类getArtist方法的调用结果。既然用MockSpotifyService模拟SpotifyService，那么模板中使用的数据无论如何都应该是mockSpotifyService的返回结果。下面一起看看如何实现。

code/routes/music/test/components/ArtistComponent.spec.ts

```
describe('renderArtist', () => {
  it('renders album info', fakeAsync(
    inject([Router, SpotifyService],
           (router: Router,
            mockSpotifyService: MockSpotifyService) => {
      const fixture = createRoot(router, RootCmp);
```

```
      let artist = {name: 'ARTIST NAME', images: [{url: 'IMAGE_1'}]};
      mockSpotifyService.setResponse(artist);

      router.navigateByUrl('/artists/2');
      advance(fixture);

      const compiled = fixture.debugElement.nativeElement;

      expect(compiled.querySelector('h1').innerHTML).toContain('ARTIST NAME');
      expect(compiled.querySelector('img').src).toContain('IMAGE_1');
    })));
  });
```

这里比较陌生的是通过mockSpotifyService的setResponse方法手动设置返回结果。

artist变量是一个测试工具夹，代表调用artists终端即使用GET方法请求https://api.spotify.com/v1/artists/{id}时从Spotify API返回的结果。

真实的JSON数据看起来如图15-1所示。

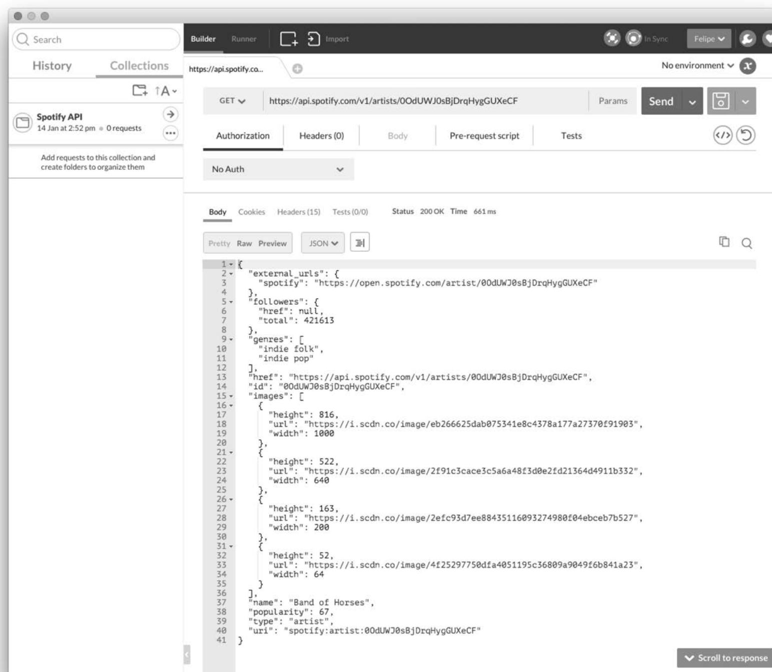


图15-1　Spotify中用来获取艺术家的服务端点

但是对于这个测试，我们仅需要name和images属性。

调用setResponse时，响应会作用于后面所有调用服务方法的下一轮调用过程。在本例中，我们要的结果是getArtist方法返回此响应。

接下来，通过路由器定位并进行advance处理。现在视图已经生成，可以使用组件视图的DOM表现形式检测是否已经正确地生成了艺术家。

fixture.debugElement.nativeElement一行中读取DebugElement的nativeElement属性可以做到这一点。

在断言语句中，我们期望H1标签中包含艺术家的名字，在本例中是字符串ARTIST NAME（源自上面的artist工具夹）。

为了检查这些条件，我们用到了NativeElement的querySelector方法。此方法会返回与CSS选择器匹配的第一个元素。

对于H1，我们检测其文本内容确实是ARTIST NAME，而图片的src属性值为IMAGE 1。

至此，我们已经完成了对ArtistComponent组件的测试。

15.10 测试表单

为了演示为表单编写测试，我们使用在第5章中创建的DemoFormNgModel组件。这个例子很不错，因为它用到了Angular表单的一些特性：

- 使用FormBuilder
- 包含表单验证
- 处理事件

下面是这个类的完整代码。

code/forms/app/forms/demo_form_with_events.ts

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  FormGroup,
  Validators,
  AbstractControl
} from '@angular/forms';

@Component({
  selector: 'demo-form-with-events',
  template: `
  <div class="ui raised segment">
    <h2 class="ui header">Demo Form: with events</h2>
    <form [formGroup]="myForm"
          (ngSubmit)="onSubmit(myForm.value)"
          class="ui form">
```

```
      <div class="field"
          [class.error]="!sku.valid && sku.touched">
        <label for="skuInput">SKU</label>
        <input type="text"
               class="form-control"
               id="skuInput"
               placeholder="SKU"
               [formControl]="sku">
        <div *ngIf="!sku.valid"
          class="ui error message">SKU is invalid</div>
        <div *ngIf="sku.hasError('required')"
          class="ui error message">SKU is required</div>
      </div>

      <div *ngIf="!myForm.valid"
        class="ui error message">Form is invalid</div>

      <button type="submit" class="ui button">Submit</button>
    </form>
  </div>
  `
})
export class DemoFormWithEvents {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];

    this.sku.valueChanges.subscribe(
      (value: string) => {
        console.log('sku changed to:', value);
      }
    );

    this.myForm.valueChanges.subscribe(
      (form: any) => {
        console.log('form changed to:', form);
      }
    );

  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form.sku);
  }
}
```

回顾一下，这段代码包含以下行为：

- 当没有值填充SKU字段时，会显示两条验证错误信息，分别是SKU is invalid和SKU is

required；

- 当SKU字段的值发生改变时，在控制台打印一条日志信息；
- 当表单发生改变时，也在控制台打印一条日志信息；
- 当提交表单时，在控制台打印最后一条日志信息。

很显然，我们用到了一个外部依赖，即控制台。正如我们之前学到的那样，必须用一些技巧来模拟所有外部依赖。

15.10.1　创建一个 ConsoleSpy

这次不用SpyObject来创建伪对象。既然我们所有使用console的场景都是调用log方法，可以让事情更简单一些。

用我们能够掌控的ConsoleSpy替换原来依赖于window.console对象的console实例。

code/forms/test/util.ts

```
export class ConsoleSpy {
  public logs: string[] = [];
  log(...args) {
    this.logs.push(args.join(' '));
  }
  warn(...args) {
    this.log(...args);
  }
}
```

ConsoleSpy会接收所有日志记录，简单地转换成字符串，并存储在其内部一个日志记录字符串列表中。



在我们的console.log版本中，为了接收可变参数，我们使用ES6和TypeScript的Rest参数[①]。
这个操作符由省略号表示，如我们的函数参数...theArgs。简单概括，使用它表示我们将接收从点号起所有剩余的参数。 例如(a, b, ...theArgs)调用了func(1, 2, 3, 4, 5)，那么a应该是1， b应该是2，而theArgs应该包含数组[3, 4, 5]。
如果你安装了最新的Node.js[②]，可以自己尝试一下：

```
$ node --harmony
> var test = (a, b, ...theArgs) => console.log('a=',a,'b=',b,'theArgs=',theArgs);
undefined
> test(1,2,3,4,5);
a= 1 b= 2 theArgs= [ 3, 4, 5 ]
```

① https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/rest_parameters
② https://nodejs.org/en/

这样，我们并不把它写进控制台本身，而是将它存储在一个数组中。如果在测试下面的代码调用了console.log三次：

```
console.log('First message', 'is', 123);
console.log('Second message');
console.log('Third message');
```

我们期望_logs字段中包含一个数组['First message is 123', 'Second message', 'Third message']。

15.10.2　安装 ConsoleSpy

为了在测试中使用探子，我们声明了两个变量：originalConsole和fakeConsole。前者存放一份原始控制台实例的引用，后者则存放控制台的模拟版本。我们还声明了一些有助于测试input和form元素的变量。

code/forms/test/forms/demo_form_with_events.spec.ts

```
describe('DemoFormWithEvents', () => {
  let originalConsole, fakeConsole;
  let el, input, form;
```

下面可以安装这个伪控制台，指定提供者。

code/forms/test/forms/demo_form_with_events.spec.ts

```
  beforeEach(() => {
    // replace the real window.console with our spy
    fakeConsole = new ConsoleSpy();
    originalConsole = window.console;
    (<any>window).console = fakeConsole;

    TestBed.configureTestingModule({
      imports: [ FormsModule, ReactiveFormsModule ],
      declarations: [ DemoFormWithEvents ]
    });
  });
```

回到测试代码，下面要做的是将真实控制台替换成我们的模板版本，换掉原始实例。

最后，在afterAll方法中将原始控制台实例还原，以免对其他测试造成泄漏（leak）。

code/forms/test/forms/demo_form_with_events.spec.ts

```
  // restores the real console
  afterAll(() => (<any>window).console = originalConsole);
```


15.10.3　配置测试模块

注意，我们在beforeEach块中调用了TestBed.configureTestingModule。要记住configure-

TestingModule方法为测试设置了根NgModule。

我们在本例中导入了两个表单模块，声明了一个DemoFormWithEvents组件。

现在我们可以操控控制台了，下面开始测试表单。

15.10.4　测试表单

现在需要对验证错误信息和表单事件进行测试。

首先要做的是取得SKU输入字段和表单元素的引用。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
it('validates and triggers events', fakeAsync((tcb) => {
  let fixture = TestBed.createComponent(DemoFormWithEvents);

  let el = fixture.debugElement.nativeElement;
  let input = fixture.debugElement.query(By.css('input')).nativeElement;
  let form = fixture.debugElement.query(By.css('form')).nativeElement;
  fixture.detectChanges();
```

最后一行通知Angular提交所有未完成的变更，正如我们在15.8节所做的那样。接下来将SKU输入值设置为空字符串。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
input.value = '';
dispatchEvent(input, 'input');
fixture.detectChanges();
tick();
```

这里我们用dispatchEvent通知Angular输入元素发生了变更，再一次触发变更检测。最后用tick()确保此时已触发的所有异步代码都已经执行。

在这个测试中使用fakeAsync和tick的目的就是为了确保表单事件被触发。如果使用async和inject替代，就必须在事件被触发前运行所有代码。

现在我们已经修改了输入值，需要确保表单验证生效。（使用el变量）查询组件元素，寻找是错误信息的所有子元素，并确保错误信息已经显示。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
let msgs = el.querySelectorAll('.ui.error.message');
expect(msgs[0].innerHTML).toContain('SKU is invalid');
expect(msgs[1].innerHTML).toContain('SKU is required');
```

接下来依葫芦画瓢，不过这次在SKU字段输入一个值。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
input.value = 'XYZ';
```

```
dispatchEvent(input, 'input');
fixture.detectChanges();
tick();
```

确保所有的错误信息消失。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
msgs = el.querySelectorAll('.ui.error.message');
expect(msgs.length).toEqual(0);
```

最后，我们触发表单的提交事件。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
fixture.detectChanges();
dispatchEvent(form, 'submit');
tick();
```

最终，我们要确保在提交表单时，通过检查打印到控制台的日志信息来确定事件被触发。

code/forms/test/forms/demo_form_with_events_bad.spec.ts

```
// checks for the form submitted message
expect(fakeConsole.logs).toContain('you submitted value: XYZ');
```

我们可以继续为另外两个事件添加新的检验：SKU变更和表单变更。然而，我们的测试代码越来越冗长。

运行测试，可以看到测试通过，如图15-2所示。

```
DemoFormWithEvents
  ✓ validates and trigger events
```

图15-2 DemoFormWithEvents测试输出

测试本身没有问题，但我们在代码风格上闻到了一些坏味道：

- 一个超长的`it`条件（超过5~10行）；
- 每个`it`中不止一两个`expect`；
- 测试描述中使用了and一词。

15.10.5 重构表单测试

解决问题的第一步就是将创建组件、获取组件元素和用于输入和表单元素的代码从中抽取出来。

code/forms/test/forms/demo_form_with_events.spec.ts

```
function createComponent(): ComponentFixture<any> {
  let fixture = TestBed.createComponent(DemoFormWithEvents);
```

```
    el = fixture.debugElement.nativeElement;
    input = fixture.debugElement.query(By.css('input')).nativeElement;
    form = fixture.debugElement.query(By.css('form')).nativeElement;
    fixture.detectChanges();

    return fixture;
  }
```

createComponent代码相当简明：使用TestBed.createComponent创建组件，获取所有元素并调用detectChanges。

现在第一个要测试的是，提供一个空的SKU字段，我们应该看到两条错误信息。

code/forms/test/forms/demo_form_with_events.spec.ts

```
  it('displays errors with no sku', fakeAsync( () => {
    let fixture = createComponent();
    input.value = '';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    // no value on sku field, all error messages are displayed
    let msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs[0].innerHTML).toContain('SKU is invalid');
    expect(msgs[1].innerHTML).toContain('SKU is required');
  }));
```

如你所见，代码清晰了很多。测试很专注，而且只测试一件事。太棒了！

在新的结构中添加第二个测试也很简单。这次要测试的是，一旦给SKU字段赋值，错误信息就会消失。

code/forms/test/forms/demo_form_with_events.spec.ts

```
  it('displays no errors when sku has a value', fakeAsync( () => {
    let fixture = createComponent();
    input.value = 'XYZ';
    dispatchEvent(input, 'input');
    fixture.detectChanges();

    let msgs = el.querySelectorAll('.ui.error.message');
    expect(msgs.length).toEqual(0);
  }));
```

你可能注意到了一点：到目前为止，我们的测试代码并没有使用fakeAsync，而是使用async和inject替代。

这次重构的另一个好处是，仅在当我们检查是否有信息发送到控制台时才使用fakeAsync和tick()，因为这正是由表单事件处理器负责的。

下一个测试恰恰是：当SKU值发生变更时，我们应该有一条信息发送到控制台。

code/forms/test/forms/demo_form_with_events.spec.ts

```
it('handles sku value changes', fakeAsync( () => {
  let fixture = createComponent();
  input.value = 'XYZ';
  dispatchEvent(input, 'input');
  tick();

  expect(fakeConsole.logs).toContain('sku changed to: XYZ');
}));
```

对于表单变更进行同样的处理。

code/forms/test/forms/demo_form_with_events.spec.ts

```
it('handles form changes', fakeAsync(() => {
  let fixture = createComponent();
  input.value = 'XYZ';
  dispatchEvent(input, 'input');
  tick();

  expect(fakeConsole.logs).toContain('form changed to: [object Object]');
}));
```

对于表单提交事件也进行同样的处理。

code/forms/test/forms/demo_form_with_events.spec.ts

```
it('handles form submission', fakeAsync((tcb) => {
  let fixture = createComponent();
  input.value = 'ABC';
  dispatchEvent(input, 'input');
  tick();

  fixture.detectChanges();
  dispatchEvent(form, 'submit');
  tick();

  expect(fakeConsole.logs).toContain('you submitted value: ABC');
}));
```

再次运行测试，会得到更清晰的输出结果，如图15-3所示。

```
DemoFormWithEvents
  ✓ displays errors with no sku
  ✓ displays no errors when sku has a value
  ✓ handles sku value changes
  ✓ handles form changes
  ✓ handles form submission
```

图15-3 重构后的DemoFormWithEvents测试输出

这次重构的另外一个好处是，出错时一眼就可以看出来。回到组件代码，在提交表单时更改消息，从而强制一个测试失败。

```
onSubmit(form: any): void {
  console.log('you have submitted the value:', form.sku);
}
```

如果运行之前版本的测试，可能看到如图15-4所示的结果。

```
DemoFormWithEvents
  ✗ validates and trigger events
      Expected [ 'sku changed to: ', 'form changed to: [object Object]', 'sku changed to: XYZ', 'form cha
nged to: [object Object]', 'you have submitted the value: XYZ' ] to contain 'you submitted value: XYZ'.
          at /Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:41894
          at run (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
          at zoneBoundFn (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
          at lib$es6$promise$$internal$$tryCatch (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.
```

图15-4　重构前的DemoFormWithEvents错误输出

它不会立即显示失败的原因所在。我们必须通过错误代号来明白提交的信息失败了。我们也不能肯定这是破坏组件的唯一事件，因为还可能有其他测试条件在遭遇失败时根本没有机会运行。

现在比较一下在重构过的代码上得到的错误信息，如图15-5所示。

```
DemoFormWithEvents
  ✓ displays errors with no sku
  ✓ displays no errors when sku has a value
  ✓ handles sku value changes
  ✓ handles form changes
  ✗ handles form submission
      Expected [ 'sku changed to: ABC', 'form changed to: [object Object]', 'you have submitted the
value: ABC' ] to contain 'you submitted value: ABC'.
          at /Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:41673
          at run (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5942)
          at zoneBoundFn (/Users/fcoury/code/ng-book2/manuscript/code/forms/test.bundle.js:5915)
          at lib$es6$promise$$internal$$tryCatch (/Users/fcoury/code/ng-book2/manuscript/code/forms/
```

图15-5　重构后的DemoFormWithEvents错误输出

这个版本很清晰，唯一失败的是表单提交事件。

15.11　测试 HTTP 请求

我们可以采用与之前相同的策略来测试HTTP交互：写一个Http类的模拟版本，因为它是一个外部依赖。

但是因为绝大多数使用Angular编写的单页面程序都是使用HTTP与API交互的，所以Angular测试类库已经提供了一个内置的替代品：MockBackend。

本章之前测试SpotifyService类时已经用到过这个类。

现在继续深入，见识一下更多的测试场景，也可以获得更好的编程实践。为了实现这个目的，我们为第6章的例子编写测试。

首先，一起看看如何测试不同的HTTP方法，如POST和DELETE，还有如何测试正要发送正确的HTTP头信息。

回到第6章，我们创建的这个实例包括了如何使用Http实现达到目的。

15.11.1 测试 POST 方法

第一个要写的测试是确保makePost方法发送一条正确的POST请求。

code/http/app/ts/components/MoreHTTPRequests.ts

```
makePost(): void {
  this.loading = true;
  this.http.post(
    'http://jsonplaceholder.typicode.com/posts',
    JSON.stringify({
      body: 'bar',
      title: 'foo',
      userId: 1
    }))
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
}
```

为这个方法编写测试时，目的是测试两点：

(1) 请求方法（POST）是正确的；

(2) 目标URL也是正确的。

下面把这个想法变成测试。

code/http/test/MoreHTTPRequests.spec.ts

```
it('performs a POST',
  async(inject([MockBackend], (backend) => {
    let fixture = TestBed.createComponent(MoreHTTPRequests);
    let comp = fixture.debugElement.componentInstance;

    backend.connections.subscribe(c => {
      expect(c.request.url)
        .toBe('http://jsonplaceholder.typicode.com/posts');
      expect(c.request.method).toBe(RequestMethod.Post);
      c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
    });

    comp.makePost();
    expect(comp.data).toEqual({'response': 'OK'});
  }))
);
```

注意，可以在`backend.connections`上调用`subscribe`方法。每当有新连接建立时就会触发我们的代码，这提供了检查请求内容的机会，并可以按预期设置响应结果。

这里，你也可以：

- 添加请求断言语句，比如检查请求的URL和HTTP方法是否正确；
- 设置模拟的响应结果，强制代码根据不同的测试场景作出不同的响应。

Angular使用一个名为`RequestMethod`的`enum`来判别不同的HTTP方法。这里是支持的方法。

```
export enum RequestMethod {
  Get,
  Post,
  Put,
  Delete,
  Options,
  Head,
  Patch
}
```

最后，在调用`makePost()`后，我们再次检查以确保预设的响应就是分配给组件的那个。

现在我们理解了其工作原理，针对`DELETE`方法增加一个测试并不难。

15.11.2　测试 DELETE 方法

这是`makeDelete`方法的具体实现。

code/http/app/ts/components/MoreHTTPRequests.ts

```
  makeDelete(): void {
    this.loading = true;
    this.http.delete('http://jsonplaceholder.typicode.com/posts/1')
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

这是我们用来测试它的代码。

code/http/test/MoreHTTPRequests.spec.ts

```
  it('performs a DELETE',
    async(inject([MockBackend], (backend) => {
      let fixture = TestBed.createComponent(MoreHTTPRequests);
      let comp = fixture.debugElement.componentInstance;

      backend.connections.subscribe(c => {
        expect(c.request.url)
          .toBe('http://jsonplaceholder.typicode.com/posts/1');
        expect(c.request.method).toBe(RequestMethod.Delete);
```

```
        c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
      });

      comp.makeDelete();
      expect(comp.data).toEqual({'response': 'OK'});
    }))
  );
```

除了URL和HTTP方法（这里使用RequestMethod.Delete）稍有不同，代码并无太大的差异。

15.11.3 测试 HTTP 头

针对这个类，最后一个要测试的方法是makeHeaders。

code/http/app/ts/components/MoreHTTPRequests.ts

```
makeHeaders(): void {
  let headers: Headers = new Headers();
  headers.append('X-API-TOKEN', 'ng-book');

  let opts: RequestOptions = new RequestOptions();
  opts.headers = headers;

  this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
    .subscribe((res: Response) => {
      this.data = res.json();
    });
}
```

在本例中，我们的测试应该集中在确保X-API-TOKEN头被正确地设置为ng-book。

code/http/test/MoreHTTPRequests.spec.ts

```
it('sends correct headers',
  async(inject([MockBackend], (backend) => {
    let fixture = TestBed.createComponent(MoreHTTPRequests);
    let comp = fixture.debugElement.componentInstance;

    backend.connections.subscribe(c => {
      expect(c.request.url)
        .toBe('http://jsonplaceholder.typicode.com/posts/1');
      expect(c.request.headers.has('X-API-TOKEN')).toBeTruthy();
      expect(c.request.headers.get('X-API-TOKEN')).toEqual('ng-book');
      c.mockRespond(new Response(<any>{body: '{"response": "OK"}'}));
    });

    comp.makeHeaders();
    expect(comp.data).toEqual({'response': 'OK'});
  }))
 );
```


请求连接的request.headers属性会返回一个Headers实例。我们使用两个方法执行两个不同的断言：

- has方法检查指定的头是否已经设置，忽略其值；
- get方法返回设置的值。

如果只检查是否设置了头即可，使用has。如果需要检测其设置的值，要使用get。

到此为止，我们完成了Angular中不同HTTP方法和头的测试。现在转向一个更为复杂的例子，它与你在编写真实程序时遇到的场景非常接近。

15.11.4 测试 **YouTubeService**

我们在第6章构建的另外一个实例是YouTube视频搜索。在这个实例中，HTTP交互过程包含在YouTubeService类中。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeService {
  constructor(private http: Http,
              @Inject(YOUTUBE_API_KEY) private apiKey: string,
              @Inject(YOUTUBE_API_URL) private apiUrl: string) {
  }

  search(query: string): Observable<SearchResult[]> {
    let params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    let queryUrl: string = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
  }
}
```

它利用YouTube API搜索视频，解析结果并保存到一个SearchResult实例中。

code/http/app/ts/components/YouTubeSearchComponent.ts

```
class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id              || null;
    this.title           = obj && obj.title           || null;
    this.description     = obj && obj.description     || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl    || null;
    this.videoUrl        = obj && obj.videoUrl        ||
                             `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

我们需要测试这个服务的几个重要方面：

- 给定一个JSON响应，服务能够分析出视频的id、标题（title）、描述（description）和缩略图（thumbnail）属性；
- 我们请求的URL使用了提供的搜索关键字；
- URL前缀设置在YOUTUBE_API_URL常量中；
- 使用的API键值与YOUTUBE_API_KEY常量匹配。

记住这些，开始编写测试。

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
  describe('MoreHTTPRequests (before)', () => {
    beforeEach(() => {
      TestBed.configureTestingModule({
        providers: [
          YouTubeService,
          BaseRequestOptions,
          MockBackend,
          { provide: YOUTUBE_API_KEY, useValue: 'YOUTUBE_API_KEY' },
          { provide: YOUTUBE_API_URL, useValue: 'YOUTUBE_API_URL' },
          { provide: Http,
            useFactory: (backend: ConnectionBackend,
                         defaultOptions: BaseRequestOptions) => {
                           return new Http(backend, defaultOptions);
                         }, deps: [MockBackend, BaseRequestOptions] }
        ]
      });

  });
```

和之前写测试的准备工作一样，首先配置依赖：这里我们使用真实的YouTubeService，但YOUTUBE_API_KEY和YOUTUBE_API_URL使用伪值。同时配置Http类使用一个MockBackend类。

现在，开始写第一个测试用例。

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
describe('search', () => {
  it('parses YouTube response',
    inject([YouTubeService, MockBackend], fakeAsync((service, backend) => {
      let res;

      backend.connections.subscribe(c => {
        c.mockRespond(new Response(<any>{
          body: `
          {
            "items": [
            {
              "id": { "videoId": "VIDEO_ID" },
              "snippet": {
                "title": "TITLE",
                "description": "DESCRIPTION",
                "thumbnails": {
                  "high": { "url": "THUMBNAIL_URL" }
                 }}}]}`
        }));
      });

      service.search('hey').subscribe(_res => {
        res = _res;
      });
      tick();

      let video = res[0];
      expect(video.id).toEqual('VIDEO_ID');
      expect(video.title).toEqual('TITLE');
      expect(video.description).toEqual('DESCRIPTION');
      expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
    }))
  )
});
```

这里我们通知Http返回一个伪响应结果，与调用真实URL时期望YouTube API返回响应结果的相关字段一致。这可以通过调用连接的mockRespond方法实现。

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
      service.search('hey').subscribe(_res => {
        res = _res;
      });
      tick();
```

接下来调用我们要测试的方法：search。调用时使用关键字hey，并抓取响应结果保存在res变量中。

你之前可能注意到了，我们使用的是fakeAsync，它需要手动调用tick()来同步异步代码。

这里沿用这种模式，期望搜索完成了执行过程，并且res已经保存了结果。

现在就可以验证结果值了。

code/http/test/YouTubeSearchComponentBefore.spec.ts

```
        let video = res[0];
        expect(video.id).toEqual('VIDEO_ID');
        expect(video.title).toEqual('TITLE');
        expect(video.description).toEqual('DESCRIPTION');
        expect(video.thumbnailUrl).toEqual('THUMBNAIL_URL');
```

从响应列表中读取第一个元素。我们已知它是SearchResult，现在要基于之前预设的响应结果检查每个属性设置正确无误：id、标题、描述和缩略图URL应该全部匹配。

至此，我们完成了写测试的第一个目标。然而，刚才不是说使用一个超大it的方法并使用太多的expect产生了代码坏味道吗？

的确是，所以在继续前进之前，先对代码进行重构，从而能更容易地使用单独的断言。

在describe('search', ...)内部添加以下辅助函数。

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
    function search(term: string, response: any, callback) {
      return inject([YouTubeService, MockBackend],
        fakeAsync((service, backend) => {
          var req;
          var res;

          backend.connections.subscribe(c => {
            req = c.request;
            c.mockRespond(new Response(<any>{body: response}));
          });

          service.search(term).subscribe(_res => {
            res = _res;
          });
          tick();

          callback(req, res);
        })
      )
    }
```

一起看看这个函数如何工作：它使用inject和fakeAsync完成了与之前同样的任务，不同的是它使用了一种配置的方式。我们提供了一个搜索关键字、一个响应和一个回调函数。有了这些参数，我们使用搜索关键字调用search方法，设置好伪响应结果，并在完成请求时调用回调函数，从而提供请求和响应对象。

使用这种方法，测试只需要调用这个函数并检查其中一个对象即可。

将之前写的测试拆分成四个测试，每个测试针对一种不同的响应结果。

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
    it('parses YouTube video id', search('hey', response, (req, res) => {
      let video = res[0];
      expect(video.id).toEqual('VIDEO_ID');
    }));

    it('parses YouTube video title', search('hey', response, (req, res) => {
      let video = res[0];
      expect(video.title).toEqual('TITLE');
    }));

    it('parses YouTube video description', search('hey', response, (req, res) =>\
{
      let video = res[0];
      expect(video.description).toEqual('DESCRIPTION');
    }));

    it('parses YouTube video thumbnail', search('hey', response, (req, res) => {
      let video = res[0];
      expect(video.description).toEqual('DESCRIPTION');
    }));
```

看起来不错吧？这个小而且专注的测试只有一个测试目的。太棒了！

现在为余下的目标添加测试代码应该很容易了。

code/http/test/YouTubeSearchComponentAfter.spec.ts

```
    it('sends the query', search('term', response, (req, res) => {
      expect(req.url).toContain('q=term');
    }));

    it('sends the API key', search('term', response, (req, res) => {
      expect(req.url).toContain('key=YOUTUBE_API_KEY');
    }));

    it('uses the provided YouTube URL', search('term', response, (req, res) => {
      expect(req.url).toMatch(/^YOUTUBE_API_URL\?/);
    }));
```

你可以按照自己的想法随意加入更多的测试。比如，针对响应结果中包含多个条目并有不同的属性添加一个测试。看看代码中是否有你想进行测试的其他方面。

15.12　总结

Angular团队在为Angular提供测试功能方面做了大量的工作。这样我们才能轻松地测试应用程序的方方面面：从控制器到服务类、表单和HTTP。本来棘手的异步代码测试现在也不费吹灰之力。

第 16 章

把AngularJS应用升级到Angular

16

如果你使用过一段时间的Angular，那么可能已经有了基于AngularJS的产品。Angular虽然很好，但我们总不能抛弃现有的一切，用Angular重写整个产品吧？更好的做法是对既有的AngularJS应用进行增量式升级。谢天谢地，Angular提供了一种非常棒的方式来实现它！

AngularJS和Angular的互操作性已经相当完善。在本章中，我们将讨论如何通过写混合式应用的方式来把AngularJS升级到Angular。这种混合式应用中同时运行着AngularJS和Angular框架（它们之间还可以交换数据）。

16.1 周边概念

当我们讨论AngularJS和Angular之间的互操作性时，会涉及很多周边概念，下面就是其中的一些。

把AngularJS的概念映射到Angular：大体上，Angular的组件就是AngularJS的指令。它们也都用到了“服务”。不过本章是讲如何同时使用AngularJS和Angular的，所以我们假设你已经充分了解了这些基础知识。如果你还没怎么用过Angular，请先阅读第3章。

把AngularJS应用迁移到Angular的准备工作：AngularJS.5提供了新的`.component`方法来制作“组件型指令”。使用`.component`有利于为迁移到Angular作准备，另外，创建瘦控制器（或禁止使用`ng-controller`指令[1]）能把AngularJS应用重构得更好，也更容易与Angular集成。

准备AngularJS应用的另一个要点是减少或消除双向绑定，更多地使用单向数据流。也就是说，尽量不要通过修改`$scope`在指令之间传递数据，而是改用服务。

这些理念确实很重要，有必要进行深入探索，但本章并不会针对升级前的重构阶段讲很多类似的最佳实践。

① http://teropa.info/blog/2014/10/24/how-ive-improved-my-angular-apps-by-banning-ng-controller.html

本章要讲的是下面这一点。

写混合式AngularJS/Angular应用：Angular提供了一种方式来启动你的AngularJS应用，然后在其中写Angular的组件和服务。写完Angular组件，只要把它和AngularJS组件混在一起就可以了。另外，依赖注入体系也支持在AngularJS和Angular之间双向传递数据，因此你写的服务在AngularJS和Angular中都能运行。

它最大的好处是什么呢？因为变更检测是在Zones中运行的，所以你再也不用调用`$scope.apply`或担心变更检测方面的问题了。

16.2　我们要构建什么

在本章中，我们准备升级一个名叫Interest的应用，它模仿了Pinterest（如图16-1所示）。其思想在于你可以保存一枚图钉（pin），即一个带图片的链接。这些图钉会显示在列表中，而且你可以收藏（或取消收藏）一枚图钉。

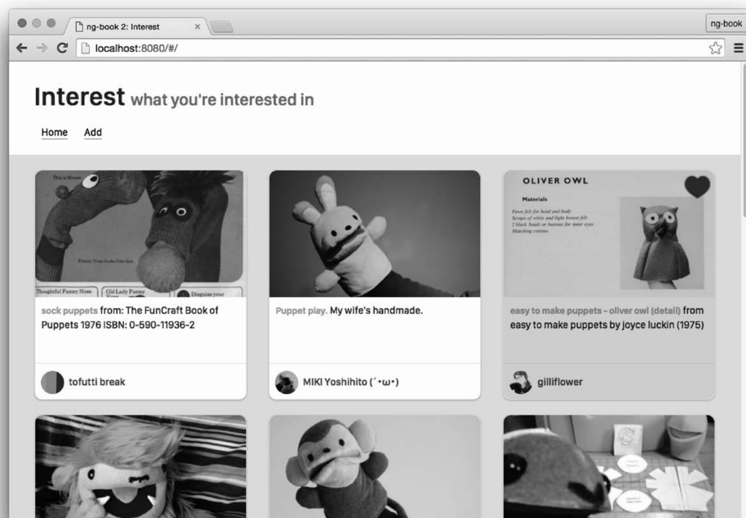


图16-1　完成后的“山寨版”Pinterest



你可以到code/conversion/AngularJS和code/conversion/hybrid下载AngularJS版和混合版的完整代码。

在深入讲解之前，我们先来看看AngularJS和Angular互操作的各种使用场景。

16.3　把 AngularJS 映射到 Angular

大体来说，AngularJS的五个主要部分是：

- 指令
- 控制器
- 作用域
- 服务
- 依赖注入

这些在Angular中则发生了显著的变化。你可能听说过，来自Angular核心团队的Igor与Tobias在2014 ngEurope大会上宣布他们将消灭AngularJS中的许多“核心”思想[①]（如图16-2所示）。具体来说，他们宣布Angular将消灭：

- `$scope`（以及默认的双向绑定）
- 指令定义对象
- 控制器
- `angular.module`



图16-2　在2014 ngEurope大会上，Igor和Tobias移除了AngularJS.x的很多API。摄影：Michael Bromley（已获授权）

① 视频地址：https://www.youtube.com/watch?v=gNmWybAyBHI

那些使用AngularJS构建应用并习惯于AngularJS思维的人可能会问：如果移除了那些，还剩下什么？没有控制器和$scope怎么能构建Angular应用呢？

尽管有很多人喜欢夸大Angular的不同之处，但其实它仍然沿袭了AngularJS的大量核心思想。事实上，Angular用一种更简单的模型实现了同样的功能。

大体上，Angular的核心构造为：

- 组件（可看作指令）
- 服务

当然，还需要大量的基础设施来支撑它们的工作。比如，你需要用依赖注入体系来管理服务；需要一个强力的变更检测机制，以便在应用中更有效地传播数据变化；还需要一个高效的渲染层，以便在正确的时机渲染DOM。

16.4　关于互操作性的需求

那么，有了这两种不同的体系，我们需要借助哪些特性来简化互操作性呢？

- 在AngularJS中使用Angular的组件：我们首先想到的是，要能写出新的Angular组件，并在AngularJS的应用中使用它们。
- 在Angular中使用AngularJS的组件：我们一般不会把整个组件树完全替换成Angular的组件，而是在Angular组件之中复用那些AngularJS组件。
- 服务共享：假设我们有一个`UserService`，想要在AngularJS和Angular之间共享它。服务通常就是一个普通的JavaScript对象，因此更抽象地说，我们需要的是一个能支持互操作的依赖注入系统。
- 变更检测：如果我们在某一边进行了改动，这些变更也应该能传播到另一边。

Angular提供了所有这些场景的解决方案，本章将一一讲解。

在本章中，我们会：

- 描述即将升级的AngularJS应用；
- 解释如何用Angular的`UpgradeAdapter`来组织混合式应用；
- 通过把AngularJS应用转化成混合式应用来一步步解释如何在AngularJS和Angular中共享组件（指令）与服务。

16.5　AngularJS 应用

作为准备，我们先重温一下该应用的AngularJS版本。



本章假设你已经具备了关于AngularJS和ui-router[①]的知识。如果你对AngularJS感到吃力，请先阅读《AngularJS权威教程》[②]。
我们不会深入剖析和解释每个AngularJS的概念，只会回顾一下这个准备升级到Angular/混合式应用的结构。

要运行AngularJS应用，使用cd转到示例代码中的conversion/AngularJS，安装依赖，并运行该应用：

```
cd code/conversion/AngularJS  # change directories
npm install                   # install dependencies
npm run go                    # run the app
```

如果没有自动打开浏览器，请手动打开URL：http://localhost:8080。

在该应用中，你可以看到用户正在收集的小玩偶。我们可以把鼠标移到某个条目上，并点击红心图标来收藏一个图钉，如图16-3所示。

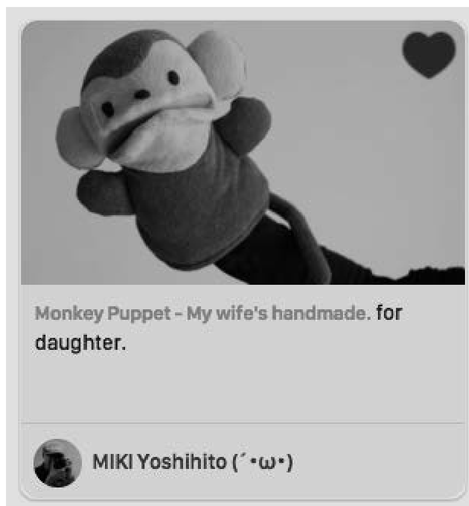


图16-3　红心表示已收藏的图钉

我们还可以导航到/add页，并添加一个新的图钉。试试提交这个默认表单。



处理图片上传对于这个演示来说过于复杂了。目前，如果你想换一幅图，只要粘贴一幅图片的完整URL即可。

① https://github.com/angular-ui/ui-router
② http://ng-book.com

16.5.1　AngularJS 应用的 HTML

AngularJS应用中的index.html使用了一种常用的结构。

code/conversion/AngularJS/index.html

```
<!DOCTYPE html>
<html ng-app='interestApp'>
<head>
  <meta charset="utf-8">
  <title>Interest</title>
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link rel="stylesheet" href="css/sf.css">
  <link rel="stylesheet" href="css/interest.css">
</head>
<body class="container-fullwidth">

  <div class="page-header">
    <div class="container">
      <h1>Interest <small>what you're interested in</small></h1>

      <div class="navLinks">
        <a ui-sref='home' id="navLinkHome">Home</a>
        <a ui-sref='add' id="navLinkAdd">Add</a>
      </div>
    </div>
  </div>

  <div id="content">
    <div ui-view=''></div>
  </div>

  <script src="js/vendor/lodash.js"></script>
  <script src="js/vendor/angular.js"></script>
  <script src="js/vendor/angular-ui-router.js"></script>
  <script src="js/app.js"></script>
</body>
</html>
```

- 注意，我们在`html`标签中使用`ng-app`来指定该应用所用的是`interestApp`模块。
- 我们在body的底部使用`script`标签来加载JavaScript脚本。
- 该模板包含一个`page-header`指令，这里是我们的导航栏。
- 我们使用了`ui-router`，这意味着：
  - 使用`ui-sref`来表示链接（Home和Add）；
  - 我们希望路由器把内容放在`ui-view`中。

16.5.2　代码概览

我们将遍历代码中的每个部分。不过首先来简单描述一下这些活动部件。

在我们的应用中，有两个路由：

- /使用HomeController；
- /add使用AddController。

我们用一个PinsService来存放所有现有图钉的数组。HomeController渲染出图钉列表，而AddController把新的元素添加到列表中。

我们的根路由使用HomeController来渲染这些图钉，而我们用pin指令来渲染单个图钉。

PinsService用于存放应用中的数据，所以先来看看它。

16.5.3　AngularJS：PinsService

code/conversion/AngularJS/js/app.js

```
angular.module('interestApp', ['ui.router'])
.service('PinsService', function($http, $q) {
  this._pins = null;

  this.pins = function() {
    var self = this;
    if(self._pins == null) {
      // initialize with sample data
      return $http.get("/js/data/sample-data.json").then(
        function(response) {
          self._pins = response.data;
          return self._pins;
        })
    } else {
      return $q.when(self._pins);
    }
  }

  this.addPin = function(newPin) {
    // adding would normally be an API request so lets mock async
    return $q.when(
      this._pins.unshift(newPin)
    );
  }
})
```

PinsService是一个.service，它把这些图钉的数组保存在属性_.pins中。

.pins方法返回一个承诺，它会被解析（resolve）成一个图钉列表。如果_.pins为null（也就是首次访问时），我们就会从/js/data/sample-data.json中加载示例数据。

code/conversion/AngularJS/js/data/sample-data.json

```
[
  {
```

```
    "title": "sock puppets",
    "description": "from:\nThe FunCraft Book of Puppets\n1976\nISBN: 0-590-11936\
-2",
    "user_name": "tofutti break",
    "avatar_src": "images/avatars/42826303@N00.jpg",
    "src": "images/pins/106033588_167d811702_o.jpg",
    "url": "https://www.flickr.com/photos/tofuttibreak/106033588/",
    "faved": false,
    "id": "106033588"
  },
  {
    "title": "Puppet play.",
    "description": "My wife's handmade.",
    "user_name": "MIKI Yoshihito (´ω)",
    "avatar_src": "images/avatars/7940758@N07.jpg",
    "src": "images/pins/4422575066_7d5c4c41e7_o.jpg",
    "url": "https://www.flickr.com/photos/mujitra/4422575066/",
    "faved": false,
    "id": "4422575066"
  },
  {
    "title": "easy to make puppets - oliver owl (detail)",
    "description": "from easy to make puppets by joyce luckin (1975)",
    "user_name": "gilliflower",
    "avatar_src": "images/avatars/26265986@N00.jpg",
    "src": "images/pins/6819859061_25d05ef2e1_o.jpg",
    "url": "https://www.flickr.com/photos/gilliflower/6819859061/",
    "faved": false,
    "id": "6819859061"
  },
```

.addPin方法把一个新图钉加入到图钉数组中。在这里，我们使用$q.when来返回一个承诺，就像我们真的向一台服务器发起异步调用时一样。

16.5.4　AngularJS：配置路由

我们准备用ui-router来配置这些路由。



如果你还不熟悉ui-router，请到https://github.com/angular-ui/ui-router/wiki阅读文档。

正如前面所说，我们有两个路由。

code/conversion/AngularJS/js/app.js

```
.config(function($stateProvider, $urlRouterProvider) {
  $stateProvider
    .state('home', {
      templateUrl: '/templates/home.html',
      controller: 'HomeController as ctrl',
```

```
      url: '/',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })
    .state('add', {
      templateUrl: '/templates/add.html',
      controller: 'AddController as ctrl',
      url: '/add',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })

    $urlRouterProvider.when('', '/') ;
})
```

第一个路由/被映射到了HomeController，我们很快就会看到它的模板。注意，我们还在使用ui-router的resolve功能。这表示在为用户加载此路由之前，我们希望先调用PinsService.pins()，并且把结果（图钉列表）注入到控制器中（HomeController）。

/add路由与之类似，只是使用了另一套模板和控制器。

我们首先看看HomeController。

16.5.5 AngularJS：HomeController

HomeController很简明。我们把通过resolve注入进来的pins保存到$scope.pins中。

code/conversion/AngularJS/js/app.js

```
.controller('HomeController', function(pins) {
  this.pins = pins;
})
```

16.5.6 AngularJS：HomeController 模板

首页的模板很小：我们用ng-repeat来循环$scope.pins中的图钉，然后用pin指令来渲染出每个图钉。

code/conversion/AngularJS/templates/home.html

```
<div class="container">
  <div class="row">
    <pin item="pin" ng-repeat="pin in ctrl.pins">
    </pin>
```

```
  </div>
</div>
```

下面深入看看这个pin指令。

16.5.7　AngularJS：pin 指令

pin指令被限制（restrict）为匹配元素（E），并且具有一个template。

我们可以通过item属性把pin传进去，就像在home.html模板中所做的那样。

link函数在定义域上定义了一个名叫toggleFav的函数，它会来回切换图钉的faved属性。

code/conversion/AngularJS/js/app.js

```
})
.directive('pin', function() {
  return {
    restrict: 'E',
    templateUrl: '/templates/pin.html',
    scope: {
      'pin': "=item"
    },
    link: function(scope, elem, attrs) {
      scope.toggleFav = function() {
        scope.pin.faved = !scope.pin.faved;
      }
    }
  }
})
```



到2016年，该指令已经不能再作为指令最佳实践的示例了。比如，要想在AngularJS中写一个全新的指令，我应该会用AngularJS.5中新的.component函数。至少，我会用controllerAs来代替link。
但本节并不是讲解该如何写好AngularJS代码的，而是展示如何迁移现有的AngularJS代码。

16.5.8　AngularJS：pin 指令模板

templates/pin.html模板在我们的页面中渲染了一个单独的图钉。

code/conversion/AngularJS/templates/pin.html

```
<div class="col-sm-6 col-md-4">
  <div class="thumbnail">
    <div class="content">
      <img ng-src="{{pin.src}}" class="img-responsive">
      <div class="caption">
```

```
        <h3>{{pin.title}}</h3>
        <p>{{pin.description | truncate:100}}</p>
      </div>
      <div class="attribution">
        <img ng-src="{{pin.avatar_src}}" class="img-circle">
        <h4>{{pin.user_name}}</h4>
      </div>
    </div>
    <div class="overlay">
      <div class="controls">
        <div class="heart">
          <a ng-click="toggleFav()">
            <img src="/images/icons/Heart-Empty.png" ng-if="!pin.faved"></img>
            <img src="/images/icons/Heart-Red.png" ng-if="pin.faved"></img>
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
```

我们在这里用到的指令都是AngularJS的内置指令：

- 用ng-src来渲染img；
- 接着显示pin.title和pin.description；
- 用ng-if来决定是显示红心还是空心。

这里最有意思的是ng-click，它会调用toggleFav，而toggleFav会修改pin.faved属性，应用从而据此显示红心或空心（如图16-4所示）。

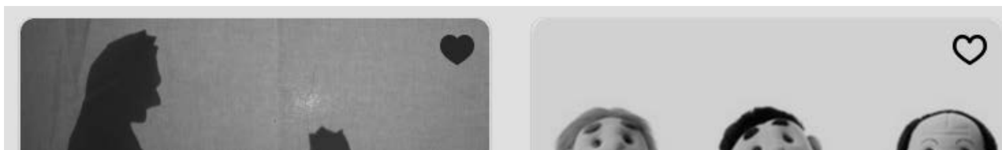


图16-4　红心与空心

接下来，我们看看AddController。

16.5.9　AngularJS：AddController

这个AddController比HomeController的代码要多一点。我们从定义控制器并指定要注入的服务开始。

code/conversion/AngularJS/js/app.js

```
.controller('AddController', function($state, PinsService, $timeout) {
  var ctrl = this;
  ctrl.saving = false;
```

16

我们在路由器和模板中使用了controllerAs语法。这意味着我们把属性放在了this上，而不是$scope上。this的作用域在ES5 JavaScript上有点复杂，所以我们指定var ctrl = this;，以消除在嵌套的函数中引用该控制器时可能出现的歧义。

code/conversion/AngularJS/js/app.js

```
  var makeNewPin = function() {
    return {
      "title": "Steampunk Cat",
      "description": "A cat wearing goggles",
      "user_name": "me",
      "avatar_src": "images/avatars/me.jpg",
      "src": "/images/pins/cat.jpg",
      "url": "http://cats.com",
      "faved": false,
      "id": Math.floor(Math.random() * 10000).toString()
    }
  }

  ctrl.newPin = makeNewPin();
```

我们创建了一个makeNewPin函数，它包含了图钉的默认构造函数和数据。

我们还通过把ctrl.newPin属性设置为该函数的调用结果初始化了该控制器。

最后，我们要定义一个函数来提交新图钉。

code/conversion/AngularJS/js/app.js

```
  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
})
```

本质上，该文档调用PinsService.addPin创建了一个新的图钉。不过这里还做了一些别的事情。

在真实的应用中，这类操作几乎总会向服务器发起一次调用。这里我们使用$timeout来模拟此效果。（实际上，你也可以移除$timeout函数，程序仍然能正常工作。在这里调用它是为了延缓程序的响应速度，让我们有机会看见Saving...提示。）

我们要给用户一些提示，好让他们知道我们正在保存图钉，因此设置ctrl.saving = true。

我们调用PinsService.addPin，并把ctrl.newPin传给它。addPin会返回一个承诺，我们在

这个承诺的回调函数中：

(1) 把ctrl.newPin恢复成原始值；

(2) 把ctrl.saving设置为false，因为已经保存好了图钉；

(3) 使用$state服务把用户重定向到首页去，在那里可以看到新的图钉。

下面是AddController的完整代码。

code/conversion/AngularJS/js/app.js

```
.controller('AddController', function($state, PinsService, $timeout) {
  var ctrl = this;
  ctrl.saving = false;

  var makeNewPin = function() {
    return {
      "title": "Steampunk Cat",
      "description": "A cat wearing goggles",
      "user_name": "me",
      "avatar_src": "images/avatars/me.jpg",
      "src": "/images/pins/cat.jpg",
      "url": "http://cats.com",
      "faved": false,
      "id": Math.floor(Math.random() * 10000).toString()
    }
  }

  ctrl.newPin = makeNewPin();

  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
})
```

16.5.10 AngularJS：AddController 模板

/add路由会渲染add.html模板。

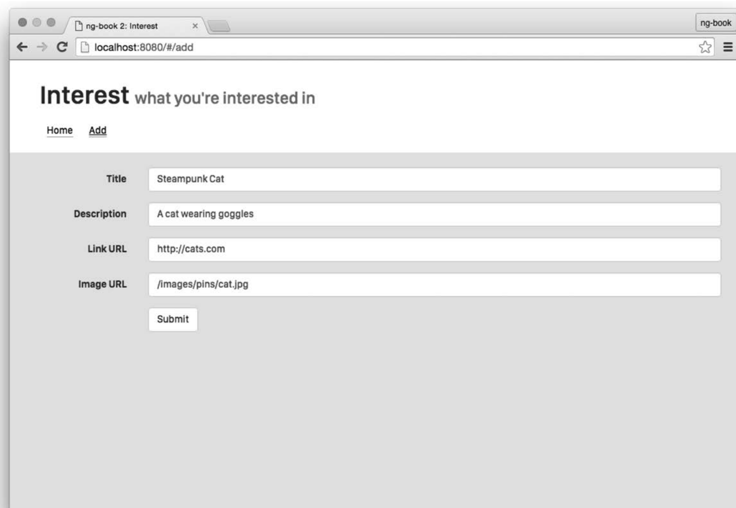


图16-5　新增图钉的表单

该模板使用ng-model来把input标签绑定到控制器上的newPin属性。

这里值得关注的是：

- 我们在提交按钮上使用ng-click来调用ctrl.submitPin；
- 如果ctrl.saving为真，那么就要显示一条Saving...消息。

code/conversion/AngularJS/templates/add.html

```
<div class="container">
  <div class="row">

    <form class="form-horizontal">

      <div class="form-group">
        <label for="title"
               class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="title"
                 placeholder="Title"
                 ng-model="ctrl.newPin.title">
      </div>
    </div>

    <div class="form-group">
      <label for="description"
             class="col-sm-2 control-label">Description</label>
      <div class="col-sm-10">
```

```
          <input type="text"
                 class="form-control"
                 id="description"
                 placeholder="Description"
                 ng-model="ctrl.newPin.description">
        </div>
      </div>

      <div class="form-group">
        <label for="url"
               class="col-sm-2 control-label">Link URL</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="url"
                 placeholder="Link URL"
                 ng-model="ctrl.newPin.url">
        </div>
      </div>

        <div class="form-group">
          <label for="url"
                 class="col-sm-2 control-label">Image URL</label>
          <div class="col-sm-10">
            <input type="text"
                   class="form-control"
                   id="url"
                   placeholder="Image URL"
                   ng-model="ctrl.newPin.src">
          </div>
        </div>

        <div class="form-group">
          <div class="col-sm-offset-2 col-sm-10">
            <button type="submit"
                    class="btn btn-default"
                    ng-click="ctrl.submitPin()">Submit</button>
          </div>
        </div>
        <div ng-if="ctrl.saving">
          Saving...
        </div>
      </form>

    </div>
  </div>
```

16.5.11 AngularJS：总结

我们终于有了要升级的AngularJS应用。该应用的复杂度正好能让我们演示如何向Angular迁移。

16.6　构建混合式应用

现在，我们已经为往现有的AngularJS应用中引入一些Angular的技术作好了准备。

开始在浏览器中使用Angular之前，我们需要对应用的结构进行一些调整。



你可以在code/conversion/hybrid找到这些示例代码。

16.6.1　混合式应用的结构

创建混合式应用的第一步是确保你同时加载了AngularJS和Angular的依赖。不过每个人遇到的具体情况可能会略有不同。

在这个例子中，我们已经提供了AngularJS的库（在js/vendor中）。接下来还要从npm中加载Angular的库。

在你的项目中，可能需要同时提供这两个库，比如使用Bower[①]等。不过对于Angular来说，用npm更省事，而且我们也建议使用npm来安装Angular。

1. 用package.json指定依赖

你可以通过npm来根据文件package.json安装依赖。下面是这个混合式应用例子中的package.json。

code/conversion/hybrid/package.json

```
{
  "name": "ng-hybrid-pinterest",
  "version": "0.0.1",
  "description": "toy pinterest clone in AngularJS/Angular hybrid",
  "contributors": [
    "Nate Murray <nate@fullstack.io>",
    "Felipe Coury <felipe@ng-book.com>"
  ],
  "main": "index.js",
  "private": true,
  "scripts": {
    "clean": "rm -f ts/*.js ts/*.js.map ts/components/*.js ts/components/*.js.ma\
p ts/services/*.js ts/services.js.map",
    "tsc": "./node_modules/.bin/tsc",
    "tsc:w": "./node_modules/.bin/tsc -w",
    "serve": "./node_modules/.bin/live-server --host=localhost --port=8080 .",
    "e2e:serve": "npm run tsc && ./node_modules/.bin/live-server --host=localhos\
t --port=8080 --no-browser .",
```

① http://bower.io/

```
    "go": "concurrent \"npm run tsc:w\" \"npm run serve\" "
  },
  "dependencies": {
    "@angular/common": "2.4.1",
    "@angular/compiler": "2.4.1",
    "@angular/core": "2.4.1",
    "@angular/forms": "2.4.1",
    "@angular/http": "2.4.1",
    "@angular/platform-browser": "2.4.1",
    "@angular/platform-browser-dynamic": "2.4.1",
    "@angular/router": "3.4.1",
    "@angular/upgrade": "2.0.0-rc.6",
    "@types/jasmine": "2.5.40",
    "core-js": "2.4.1",
    "es6-shim": "0.35.0",
    "reflect-metadata": "0.1.9",
    "rxjs": "5.0.2",
    "systemjs": "0.19.6",
    "ts-helpers": "1.1.1",
    "tslint": "3.7.0-dev.2",
    "typings": "0.8.1",
    "zone.js": "0.7.4"
  },
  "devDependencies": {
    "@types/jasmine": "2.2.30",
    "@types/node": "6.0.42",
    "concurrently": "1.0.0",
    "jasmine-spec-reporter": "2.5.0",
    "karma": "0.12.22",
    "karma-chrome-launcher": "0.1.4",
    "karma-jasmine": "0.1.5",
    "live-server": "0.9.0",
    "protractor": "4.0.14",
    "ts-node": "1.2.1",
    "typescript": "2.0.3"
  }
}
```



如果你不熟悉其中的某个包，最好自己去发现它的用途。比如rxjs是一个为我们提供可观察对象的库，而systemjs提供的是模块加载器，我们将在本章中用到它。

一旦添加了Angular的依赖，就可以运行npm install命令来安装它们了。

2. 编译代码

你可能注意到了，package.json中的"script"属性中包含另一个属性"tsc"。这表示当我们运行命令npm run tsc时，它就会调用TypeScript编译器来编译我们的代码。

我们准备在这个例子中使用TypeScript，同时AngularJS的代码仍然使用JavaScript。

要这么做，就要先把所有TypeScript代码放进ts/文件夹里，把所有JavaScript代码放进js/文件夹里。

我们用tsconfig.json文件来配置TypeScript编译器。关于此文件，现在你只要知道一点就可以了：`filesGlob`属性指定了适配规则`"./ts/**/*.ts"`。它的意思是“当运行TypeScript编译器时，我们希望编译ts/目录下所有以.ts结尾的文件”。

在该项目中，浏览器只会加载JavaScript。因此我们要使用TypeScript编译器（`tsc`）来把这些代码编译成JavaScript，然后再把AngularJS和Angular的JavaScript代码加载进浏览器中。

3. 加载index.html依赖

现在，我们已经设置好了依赖和编译器，接着就要把这些JavaScript文件加载到浏览器中了。因此，我们添加`script`标签。

code/conversion/AngularJS/hybrid/index.html

```
<div id="content">
  <div ui-view=''></div>
</div>

<!-- Libraries -->
<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/reflect-metadata/Reflect.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>

<script src="js/vendor/angular.js"></script>
<script src="js/vendor/angular-ui-router.js"></script>
```

我们从node_modules/中加载的文件是Angular及其依赖，而从js/vendor/中加载的文件则是AngularJS及其依赖。

但是你可能已经注意到了，我们还没有在HTML标签中加载任何自己的代码。要加载这些代码，就要使用System.js。

4. 配置System.js

在这个例子中，我们准备把System.js用作模块加载器。



我们还可以使用webpack（就像在本书其他例子中所用的那样）或很多其他的加载器（比如requirejs等）。不过，System.js是一个很不错并且具有可伸缩性的加载器，常常和Angular一起使用。本章会提供一个漂亮的示例，向你展示如何通过System.js使用Angular。

要配置System.js，需要在index.html的`<script>`标签中进行如下修改：

```
<script src="resources/systemjs.config.js"></script>
System.import('ts/app.js')
      .then(null, console.error.bind(console));
```

System.import('ts/app.js')说明该应用的入口点是ts/app.js文件。当我们写混合式Angular应用时，Angular的代码会成为入口点。这很容易理解，因为Angular提供了对AngularJS的向后兼容能力。我们很快就会看到如何引导该应用。

这里要注意的另一个问题是，我们正在ts/目录下加载.js文件。为什么呢？这是因为TypeScript编译器会在页面加载时把这些文件编译成JavaScript。

我们已经在resources/systemjs.config.js中配置好了System.js。此文件中包含了几乎标准化的配置方式，但现在我们要把AngularJS应用加载到Angular代码中，那就不得不添加一个特殊的属性interestAppNg1了，它指向我们的AngularJS应用。该选项让我们能在TypeScript代码中这样用：

```
import 'interestAppNg1'; // "bare import" for side-effects
```

当模块加载器看到字符串'interestAppNg1'时，就会去./js/app.js中加载我们的AngularJS应用。

packages属性指出ts包（package）中的文件将会具有.js扩展名，并使用System.js来注册（register）这种模块格式。



TypeScript编译器可以输出多种模块格式。System.js的format需要与编译器输出的模块格式保持一致。这里register的模块格式之所以能直接使用，是因为我们在tsconfig.json中把compilerOptions.module指定成了"system"格式。



要配置好System.js是很难的，有大量潜在选项。
这不是一本关于模块加载器的书，事实上，只是深入讲解如何配置System.js和其他JavaScript模块加载器就足够写一整本书了。
目前，我们不准备深入讨论模块加载器，不过如果你想了解更多，请参阅https://github.com/systemjs/systemjs/blob/master/docs/config-api.md。



你想阅读关于JavaScript模块加载器的书吗？我们正在考虑写一本。如果你想及时收到通知，请在这里留下你的邮箱：http://eepurl.com/bMOaEX。

16.6.2　引导混合式应用

现在项目结构已经就绪，我们来启动这个应用吧。

还记得吗？在AngularJS中，有两种方式可以启动应用：

(1) 使用ng-app指令，比如在HTML中写ng-app='interestApp'；

(2) 在JavaScript中使用angular.bootstrap。

在混合式应用中，我们要使用来自UpgradeAdapter的新引导方法。

我们还要改为从代码中启动应用，因此请确保从index.html中移除了ng-app指令。

一个最简的启动代码是这样的：

```
// code/conversion/hybrid/ts/app.ts
import {
  NgModule,
  forwardRef
} from '@angular/core';
import { CommonModule } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';

import { UpgradeAdapter } from '@angular/upgrade';
declare var angular: any;
import 'interestAppNg1'; // "bare import" for side-effects

/*
 * Create our upgradeAdapter
 */
const upgradeAdapter: UpgradeAdapter = new UpgradeAdapter(
  forwardRef(() => MyAppModule)); // <-- notice forward reference

// ...
// upgrade and downgrade components in here
// ...

/*
 * Create our app's entry NgModule
 */
@NgModule({
  declarations: [ MyNg2Component, ... ],
  imports: [
    CommonModule,
    BrowserModule
  ],
  providers: [ MyNg2Services, ... ]
})
class MyAppModule { }

/*
 * Bootstrap the App
 */
upgradeAdapter.bootstrap(document.body, ['interestApp']);
```

我们先导入了UpgradeAdapter，然后创建它的实例upgradeAdapter。

不过，UpgradeAdapter的构造函数需要一个NgModule，它用于启动我们的Angular应用，但我们还没有定义它呢！要解决这个问题，就用forwardRef函数来取得NgModule的“前向引用”（后面会声明它）。

当我们定义自己的NgModule也就是MyAppModule时（具体到这个应用中，它应该是InterestAppModule），写法和定义其他Angular的NgModule没有区别：我们放进了声明（declarations）、导入（imports）和提供者（providers）等。

最后，我们告诉upgradeAdapter在document.body元素上bootstrap此应用，并指定了AngularJS应用的模块名。

这将会在启动Angular应用的同时启动AngularJS应用！接下来，我们就开始一点一点地用Angular替换掉它。

16.6.3　我们要升级什么

先来讨论一下这个例子中的哪些部分需要迁移到Angular，哪些仍然留在AngularJS。

1. 首页

需要注意的第一点是，我们仍将使用AngularJS来管理路由。当然，Angular有自己的路由，你可以在第7章中读到它。但是如果你正在构建一个混合式应用，很可能已经用AngularJS配置过很多路由了。因此，在这个例子中，我们仍然沿用ui-router作为路由体系。

在首页中，我们准备把Angular的组件嵌套在AngularJS的指令中。在这个例子中，就是把“图钉控件”转变成Angular的组件（如图16-6所示）。也就是说，我们的pin指令将调用Angular的pin-controls组件，而pin-controls组件负责渲染出用来表示收藏的心型图标。

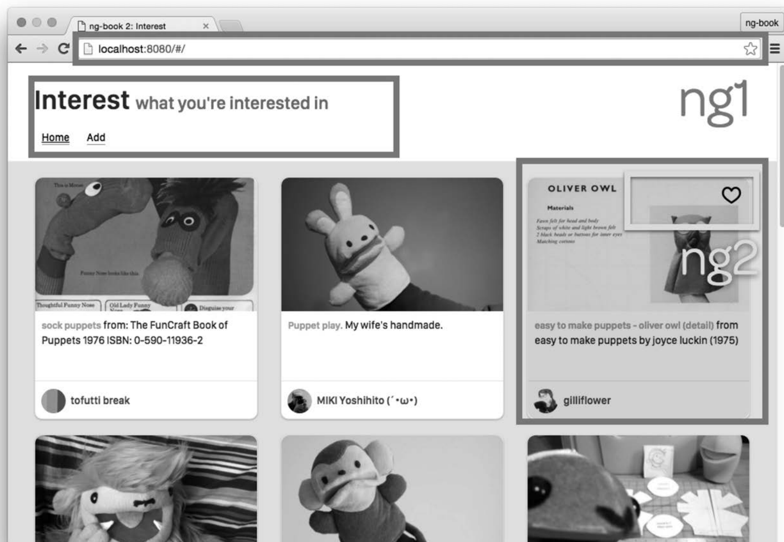


图16-6　首页的AngularJS和Angular组件

尽管这是一个很小的例子，但它展示了一种强有力的想法：如何在ng的不同版本之间无缝地交换数据。

2. About页

我们也会在About页上使用AngularJS来实现路由和页眉。不过，在About页上，我们将把整个表单替换成Angular的组件：`AddPinComponent`（如图16-7所示）。

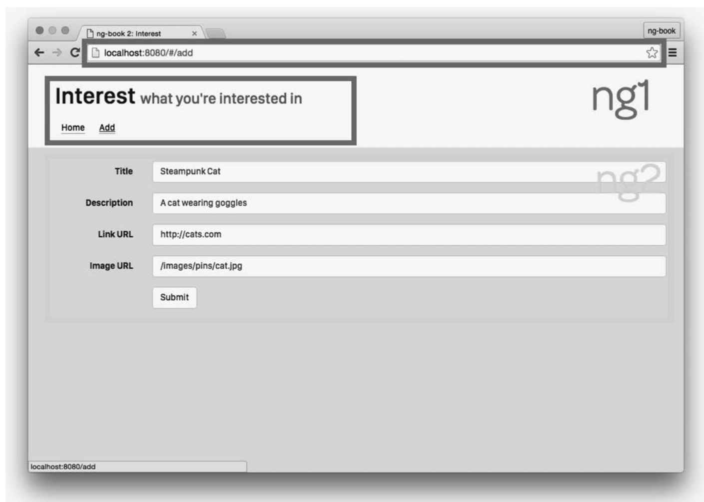


图16-7　About页的AngularJS和Angular组件

回想一下，该表单会往`PinsService`上添加一个新的图钉。在这个例子中，我们需要通过某种方式来让Angular的`AddPinComponent`访问到AngularJS的`PinsService`。

另外，在添加新的图钉之后，该应用应该自动导航到首页。不过，要想改变当前路由，我们需要在Angular的`AddPinComponent`中使用来自AngularJS中`ui-router`库的`$state`服务。因此，我们同样需要确保`$state`服务也能在`AddPinComponent`中使用。

3. 服务

我们刚才说过，有两个AngularJS的服务将会升级到Angular：

- `PinsService`
- `$state`

不过我们也想看看如何把一个Angular服务降级，以供AngularJS使用。为此，我们稍后会用TypeScript/Angular来创建一个`AnalyticsService`服务，并把它共享给AngularJS。

4. 盘点

概括起来，我们准备讲解下列内容：

- 把Angular的PinControlsComponent降级到AngularJS（用来实现收藏按钮）；
- 把Angular的AddPinComponent降级到AngularJS（用来实现新增图钉页面）；
- 把Angular的AnalyticsService降级到AngularJS（用来进行事件记录）；
- 把AngularJS的PinsService升级到Angular（用来新增图钉）；
- 把AngularJS的$state服务升级到Angular（用来控制路由）。

16.6.4 插一小段内容：类型文件

TypeScript最美妙的一点就是编译时类型检查。不过，如果你正在构建一个混合式应用，那么估计你打算集成到项目中的JavaScript代码大部分是无类型的。

当你试图在TypeScript中使用JavaScript代码时，可能会收到编译器错误，因为编译器不知道你的JavaScript对象结构如何。你可以尝试把它们全部转换成<any>，但这样不但看起很来丑而且容易出错。

更好的方案是给TypeScript编译器提供自定义*类型注解*。然后，编译器就能用这些类型信息来强化你的JavaScript代码了。

比如，还记得我们是怎样在AngularJS版本的makeNewPin中创建图钉对象的吗？

code/conversion/AngularJS/js/app.js

```
var makeNewPin = function() {
  return {
    "title": "Steampunk Cat",
    "description": "A cat wearing goggles",
    "user_name": "me",
    "avatar_src": "images/avatars/me.jpg",
    "src": "/images/pins/cat.jpg",
    "url": "http://cats.com",
    "faved": false,
    "id": Math.floor(Math.random() * 10000).toString()
  }
}

ctrl.newPin = makeNewPin();
```

如果能把这些对象的结构告诉编译器该多好！那样就不用到处求助于any了。

此外，我们准备在Angular/TypeScript中使用ui-router中的$state服务，同样要把这个服务中有哪些可用的函数告诉编译器。

因此，虽然为TypeScript提供自定义类型信息是TypeScript的分内之事（与Angular无关），但我们还是得亲力亲为。现在之所以还缺少这么多类型定义文件，是因为TypeScript才发布没多久，仍然相对较新。

在本节中，我会告诉你如何为TypeScript制作自定义类型文件（custom typing）。



如果你已经很熟悉如何创建和使用TypeScript的类型定义文件，请放心大胆地跳过本节。

1. 类型文件

在TypeScript中，可以通过书写类型定义文件（typing definition file）来描述我们的代码结构。类型定义文件通常以扩展名.d.ts结尾。

当我们写TypeScript代码时，通常不用写.d.ts文件，因为TypeScript文件本身已经包含了类型信息。只有当要为某些外来的JavaScript代码添加类型信息时，才需要写.d.ts文件。

例如，为了描述我们的图钉对象，可以为它写一个`interface`。

code/conversion/hybrid/js/app.d.ts

```
export interface Pin {
  title: string;
  description: string;
  user_name: string;
  avatar_src: string;
  src: string;
  url: string;
  faved: boolean;
  id: string;
}
```

注意，我们不是在声明一个类，也没有创建实例，而是定义了接口的形态（类型）。

要使用.d.ts文件，需要告诉TypeScript它们在哪里。最简单的方式就是修改tsconfig.json文件。比如，假设有一个名为js/app.d.ts的文件，我们就可以像这样添加它：

```
// tsconfig.json
"compilerOptions": { ... },
"files": [
  "ts/app.ts",
  "js/app.d.ts"
],
// more...
```

仔细看这里的文件路径。我们要从ts/app.ts中加载TypeScript，从js/目录下加载app.d.ts文件。这是因为js/app.d.ts文件是为js/app.js（这是AngularJS的JavaScript文件，而不是Angular的TypeScript文件）准备的类型文件。

我们这就一点点把app.d.ts写出来。首先来看一个现有工具`typings`，以帮助我们使用第三方TypeScript定义文件。

2. 使用`typings`管理第三方库

`typings`是一个用来为第三方库管理TypeScript类型定义文件的工具。

我们准备使用angular-ui-router，所以要用typings来安装angular-ui-router的类型信息。下面是操作步骤。

先安装好typings，可以用命令npm install -g typings来安装。

接下来，配置一个typings.json文件，可以用命令typings init来创建（或者使用现成的）。

然后，我们通过命令typings install angular-ui-router --save来安装所需的包。

注意，typings命令创建了一个typings目录，其中包含文件browser.d.ts。这个browser.d.ts文件是所有被typings管理的类型定义文件的总入口点。也就是说，如果你写了自己的类型定义文件，那么它们不会被包含在这里，但通过typings工具安装的类型定义都会被加载到此文件的reference标签下。



不要直接修改typings/browser.d.ts文件！typings会替你管理这个文件。如果你修改了它，那么这些修改就会被覆盖。

现在，我们有了类型定义文件typings/browser.d.ts，该如何使用它呢？我们得先把它告诉编译器才行。可以通过tsconfig.json来做到这一点：

```
// tsconfig.json
"compilerOptions": { ... },
"files": [
  "typings/browser.d.ts",
  "ts/app.ts",
  "js/app.d.ts"
],
// more...
```

注意，我们把typings/browser.d.ts文件添加到了files数组中。这会告诉编译器我们要在编译时包含typings下的类型信息。



假如我们要加载另一个库(比如underscore),而且同样希望用System.js加载它，该怎么办呢？

整体思路是，你要：(1) 让类型信息在编译时可用；(2) 让代码在运行时可用。具体办法如下。

(1) typings install underscore：安装类型信息文件。

(2) npm install underscore：在node_modules中安装JavaScript文件。

(3) 在index.html中调用System.config的地方往paths下增加一句underscore: './node_modules/underscore/underscore.js'。

(4) 然后在TypeScript中通过import * as _ from 'underscore';导入下划线。

(5) 最后使用下划线，就像这样：let foo = _.map([1,2,3], (x) => x + 1);。



我们已经在这个应用中做完了typings install，所以你不必自己安装这些依赖了。

实际上，如果你运行typings install，将会收到一个错误：

```
node_modules/angular2/typings/angular-protractor/angular-protractor.d.ts(1679,13\
): error TS2403: Subsequent variable declarations must have the same type.  Vari\
able '$' must be of type 'JQueryStatic', but here has type 'cssSelectorHelper'.
```

这个bug是因为jquery和angular的类型信息都试图把一个类型赋给$变量。在本书出版时，临时性的解决方案是打开typings/jquery/jquery.d.ts文件，并注释掉这一行：

```
// declare var $: JQueryStatic; // - ng-book told me to comment this
```

当然，如果你想在TypeScript中通过$来访问jQuery特有的类型信息，就会出错（不过本例中不存在这种情况）。

3. 自定义类型文件

能使用现成的第三方类型定义文件固然好，不过还有一些场景是找不到现有类型定义文件的，特别是我们自己写的代码。

通常，当我们写自定义类型信息文件时，会把它和相应的JavaScript代码放在一起，因此我们来创建一个js/app.d.ts文件。

code/conversion/hybrid/js/app.d.ts

```
declare module interestAppNg1 {

  export interface Pin {
    title: string;
    description: string;
    user_name: string;
    avatar_src: string;
    src: string;
    url: string;
    faved: boolean;
    id: string;
  }

  export interface PinsService {
    pins(): Promise<Pin[]>;
    addPin(pin: Pin): Promise<any>;
  }

}

declare module 'interestAppNg1' {
  export = interestAppNg1;
}
```

我们用`declare`关键字来制作“周边声明”（ambient declaration），意思是我们定义了一个并非来自于TypeScript文件的变量。在这个例子中，我们声明了两个接口：

(1) `Pin`

(2) `PinsService`

`Pin`接口用来描述图钉对象的属性名及其值类型。

`PinsService`接口则用来描述这个`PinsService`中两个方法的类型。

- `pins()`返回一个由`Pin`数组构成的`Promise`；
- `addPin()`接收一个`Pin`参数，并返回一个`Promise`。



学习写类型定义文件的更多知识

如果要学习关于写.d.ts文件的更多知识，下列链接会很有帮助：

- “TypeScript手册：与其他JavaScript库协同工作”（http://www.typescriptlang.org/Handbook#modules-working-with-other-javascript-libraries）
- “TypeScript手册：书写类型定义文件”（https://github.com/Microsoft/TypeScript-Handbook/blob/master/pages/Writing%20Definition%20Files.md）
- “快速提示：TypeScript的`declare`关键字”（http://blogs.microsoft.co.il/gilf/2013/07/22/quick-tip-typescript-declare-keyword/）

你可能已经注意到了，我们并没有在AngularJS的JavaScript代码的任何地方声明令牌`interestAppNg1`。这是因为`interestAppNg1`只是我们用来在TypeScript代码中引用这些JavaScript代码时所用的标识符，并不是类型的一部分。

我们已经完成了这个文件，可以导入这些类型了，就像这样：

```
import { Pin, PinsService } from 'interestAppNg1';
```

16.6.5 写 Angular 的 `PinControlsComponent`

我们刚刚明白了类型信息，那就言归正传，继续看混合式应用吧。

我们首先要做的是写一个Angular版的`PinControlsComponent`，这样才能把Angular的组件嵌入到AngularJS的指令中。`PinControlsComponent`为收藏的图钉显示心型图标，点击它就可以来回切换状态。

先从导入`Pin`类型开始，然后定义另一些需要的常量。

code/conversion/hybrid/ts/components/PinControlsComponent.ts

```
/*
 * PinControls: a component that holds the controls for a particular pin
```

```
 */
import {
  Component,
  Input,
  Output,
  EventEmitter
} from '@angular/core';
import { NgIf } from '@angular/common';
import { Pin } from 'interestAppNg1';
```

接下来写@Component注解。

code/conversion/hybrid/ts/components/PinControlsComponent.ts

```
@Component({
  selector: 'pin-controls',
  template: `
<div class="controls">
  <div class="heart">
    <a (click)="toggleFav()">
      <img src="/images/icons/Heart-Empty.png" *ngIf="!pin.faved" />
      <img src="/images/icons/Heart-Red.png" *ngIf="pin.faved" />
    </a>
  </div>
</div>
  `
})
```

注意，这里匹配的是pin-controls元素。

我们的模板和AngularJS版本的很像，只是把(click)和*ngIf改成了用Angular的模板语法。

现在的组件定义类变成了下面这样。

code/conversion/hybrid/ts/components/PinControlsComponent.ts

```
export class PinControlsComponent {
  @Input() pin: Pin;
  @Output() faved: EventEmitter<Pin> = new EventEmitter<Pin>();

  toggleFav(): void {
    this.faved.next(this.pin);
  }
}
```

注意，我们并没有在@Component注解中指定inputs和outputs，而是直接在类的属性上使用了@Input和@Output注解。用这种方式为属性提供类型信息更加简便。

该组件将接收一个pin参数作为输入，也就是我们管理的Pin对象。

该组件指定了一个名叫faved的输出参数。这跟我们在AngularJS应用中的用法略有不同。如果你查看toggleFav的实现，会发现我们所做的是通过EventEmitter把当前图钉发给了外界。

这是因为我们已经在AngularJS中实现了更改faved状态的方法，所以不希望在Angular中重新实现一模一样的功能（但你也可能希望再次实现，这取决于你们开发组内的约定）。

16.6.6　使用 Angular 的 PinControlsComponent

有了Angular的pin-controlls组件，我们就可以在模板中使用它了。现在的pin.html模板变成了下面这样。

code/conversion/hybrid/templates/pin.html

```
<div class="col-sm-6 col-md-4">
  <div class="thumbnail">
    <div class="content">
      <img ng-src="{{pin.src}}" class="img-responsive">
      <div class="caption">
        <h3>{{pin.title}}</h3>
        <p>{{pin.description | truncate:100}}</p>
      </div>
      <div class="attribution">
        <img ng-src="{{pin.avatar_src}}" class="img-circle">
        <h4>{{pin.user_name}}</h4>
      </div>
    </div>
    <div class="overlay">
      <pin-controls [pin]="pin"
                    (faved)="toggleFav($event)"></pin-controls>
    </div>
  </div>
</div>
```

该模板是属于AngularJS指令的，因此我们可以在里面使用AngularJS的指令，比如ng-src。不过，要注意使用Angular中pin-controls组件的那一行：

```
<pin-controls [pin]="pin"
              (faved)="toggleFav($event)"></pin-controls>
```

有意思的是，我们在同时使用Angular输入属性的方括号语法[pin]以及Angular输出属性的圆括号语法(faved)。

在混合式应用中，当你在AngularJS中使用Angular指令时，仍然可以照常使用Angular的语法。

通过输入属性[pin]，可以把来自AngularJS指令scope上的pin属性传进去。

在输出参数(faved)中，我们调用了AngularJS指令scope上的toggleFav函数。注意看这里的实现方式：我们没有在Angular指令中修改pin.faved状态（虽然我们也能这么做）；反之，我们只是让Angular的PinControlsComponent在调用toggleFav的时候把这个pin发给外界。（如果没看明白，请再回头看看PinControlsComponent的toggleFav。）

我们这么做是为了告诉你：可以保持AngularJS中的现有功能（scope.toggleFav）不变，只把组件迁移到Angular。在这个例子中，AngularJS的pin指令监听了Angular PinControls-Component上的faved事件。

如果你刷新这个页面，可能会注意到它无法正常工作，那是因为我们还缺少一个很重要的步骤：把`PinControlsComponent`降级到AngularJS。

16.6.7　把 Angular 的 `PinControlsComponent` 降级到 AngularJS

要想让我们的组件跨越Angular和AngularJS的界线，最后一步是使用`upgradeAdapter`来降级这些组件（或者升级，我们稍后会看到）。

我们在app.ts文件中执行这些降级工作（也就是调用`upgradeAdapter.bootstrap`的地方）。

首先，我们需要导入必备的`angular`库。

code/conversion/hybrid/ts/app.ts

```
import {
  NgModule,
  forwardRef
} from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  FormsModule,
} from '@angular/forms';
import { BrowserModule } from "@angular/platform-browser";
import { UpgradeAdapter } from '@angular/upgrade';
declare var angular: any;
import 'interestAppNg1'; // "bare import" for side-effects
```

然后，我们用（几乎）标准的AngularJS方式来创建一个`.directive`。

code/conversion/hybrid/ts/app.ts

```
angular.module('interestApp')
  .directive('pinControls',
           upgradeAdapter.downgradeNg2Component(PinControlsComponent))
```

记住我们已经导入了`'interestAppNg1'`，它会加载我们的AngularJS应用，而AngularJS应用中调用了`angular.module('interestApp', [])`。也就是说，我们的AngularJS应用已经通过`angular`注册好了`interestApp`模块。

现在，我们要通过调用`angular.module('interestApp')`来找到该模块，然后把指令添加到其中，就像我们在AngularJS中的标准做法那样。



`angular.module`的获取（getter）和设置（setter）语法

还记得吗？当往`angular.module`函数的第二个参数中传入一个数组时，我们就是在**创建**模块。比如`angular.module('foo', [])`将创建一个名叫`foo`的模块。我们非正式地将其称为设置语法。

同样，如果我们省略了这个数组，就是在**获取**一个模块（假设它已经存在）。比如`angular.module('foo')`将获取`foo`模块。我们称其为获取语法。



在这个例子中，如果我们忘了这项限制，并且在app.ts（Angular）中调用`angular.module('interestApp', [])`，就会意外地覆盖现有的interestApp模块。你的应用将无法正常工作。千万要小心！

我们调用`.directive`并创建了一个名叫`'pinControls'`的指令。这是一种标准的AngularJS实践。它的第二个参数是指令定义对象（directive definition object，DDO），我们不会手动创建DDO，而是调用`upgradeAdapter.downgradeNg2Component`。

`downgradeNg2Component`会把我们的`PinControlsComponent`转换成与AngularJS兼容的指令。干净！漂亮！

刷新一下，你会发现收藏功能仍然正常工作（如图16-8所示），但我们已经把目前的实现方式改成在AngularJS中嵌入Angular了！



图16-8　收藏功能仍然很棒

16.6.8　用 Angular 添加图钉

接下来要用Angular组件对添加图钉的页面进行升级（如图16-9所示）。

图16-9　新增图钉的表单

回想一下，这个页面一共做了三件事：

(1) 为用户提供一个用来描述这个图钉的表单；

(2) 借助PinsService把新的图钉添加到图钉列表中；

(3) 把用户重定向到首页。

我们来看看该如何在Angular中做到这些。

Angular提供了一个强力的表单库，所以这没什么难度。那我们就来写一个正统的Angular表单吧。

不过，PinsService仍然来自AngularJS。通常，我们会有很多来自AngularJS的既有服务，但又没那么多时间把它们都改写成Angular的。因此在这个例子中，我们仍然把PinsService保留为AngularJS对象，并把它注入到Angular中。

与之类似，我们把来自AngularJS的ui-router作为路由系统。要想在ui-router中进行页面跳转，就得使用$state服务，它是一个AngularJS服务。

那么，在这里要做的就是把PinsService和$state服务从AngularJS升级到Angular，这已经是最简易的方式了。

16.6.9　把 AngularJS 的 PinsService 和$state 升级到 Angular

要想升级AngularJS的服务，我们可以调用upgradeAdapter.upgradeNg1Provider。

code/conversion/hybrid/ts/app.ts

```
/*
 * Expose our AngularJS content to Angular
*/
upgradeAdapter.upgradeNg1Provider('PinsService');
upgradeAdapter.upgradeNg1Provider('$state');
```

这样就足够了。现在我们可以把AngularJS的服务注入（@Inject）到Angular的组件中，就像这样：

```
class AddPinComponent {
  constructor(@Inject('PinsService') public pinsService: PinsService,
              @Inject('$state') public uiState: IStateService) {
  }
  // ...
  // now you can use this.pinsService
  // or this.uiState
  // ...
}
```

在这个构造函数中，有几点需要注意。

@Inject注解的意思是，我们要把参数中指定的可注入对象解析出来，赋值给紧随其后的变量。比如这里的pinsService将被赋值为我们在AngularJS中定义的服务PinsService。

在TypeScript语法中，在constructor中使用public关键字其实是一种简写形式，用来把该变量赋值给this。也就是说，当我们写public pinsService时，其实是在做两件事：(1) 在该类上定义一个实例属性pinsService；(2) 把构造函数的参数pinsService赋值给this.pinsService。

最终的效果是我们可以在这个类中访问this.pinsService了。

最后，我们定义了所注入的两个服务的类型：PinsService和IStateService。

PinsService来自我们以前定义过的app.d.ts。

code/conversion/hybrid/js/app.d.ts

```
export interface PinsService {
  pins(): Promise<Pin[]>;
  addPin(pin: Pin): Promise<any>;
}
```

IStateService来自ui-router的类型文件，它是我们以前用typings工具安装的。

通过把这些服务的类型信息告诉TypeScript，我们在写代码时就可以享受类型检查带来的好处了。

下面来写完AddPinComponent的剩余部分。

16.6.10 写 Angular 版的 AddPinComponent

我们先从导入所需的类型信息开始。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
/*
 * AddPinComponent: a component that controls the "add pin" page
 */
import {
  Component,
  Inject
} from '@angular/core';
import { Pin, PinsService } from 'interestAppNg1';
import { IStateService } from 'angular-ui-router';
```

注意，我们导入了自定义类型Pin和PinsService，还从angular-ui-router中导入了IStateService。

1. AddPinComponent的@Component

这个@Component注解非常简明。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
@Component({
  selector: 'add-pin',
  templateUrl: '/templates/add-Angular.html'
})
```

2. AddPinComponent模板

我们使用templateUrl来加载模板。在该模板中，我们的表单和AngularJS中的表单非常像，但所用的是Angular的表单指令集。



我们在这里不准备深入讲解ngModel/ngSubmit。如果你想深入了解Angular表单的工作原理，请阅读第5章，我们在那里对表单进行了详细讲解。

code/conversion/hybrid/templates/add-Angular.html

```
<div class="container">
  <div class="row">

    <form (ngSubmit)="onSubmit()"
          class="form-horizontal">

      <div class="form-group">
        <label for="title"
               class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text"
                 class="form-control"
                 id="title"
                 name="title"
                 placeholder="Title"
                 [(ngModel)]="newPin.title">
        </div>
```

这里使用了两个指令：ngSubmit和ngModel。

我们在表单上使用了(ngSubmit)。这样当表单被提交时，就会调用onSubmit函数。（我们会在稍后的AddPinComponent控制器中定义onSubmit函数。）

我们使用[(ngModel)]来把title输入框的值绑定到控制器中newPin.title的值。

下面是完整的模板代码。

code/conversion/hybrid/templates/add-Angular.html

```
<div class="container">
  <div class="row">

    <form (ngSubmit)="onSubmit()"
          class="form-horizontal">
```

```
<div class="form-group">
  <label for="title"
         class="col-sm-2 control-label">Title</label>
  <div class="col-sm-10">
    <input type="text"
           class="form-control"
           id="title"
           name="title"
           placeholder="Title"
           [(ngModel)]="newPin.title">
  </div>
</div>

<div class="form-group">
  <label for="description"
         class="col-sm-2 control-label">Description</label>
  <div class="col-sm-10">
    <input type="text"
           class="form-control"
           id="description"
           name="description"
           placeholder="Description"
           [(ngModel)]="newPin.description">
  </div>
</div>

<div class="form-group">
  <label for="url"
         class="col-sm-2 control-label">Link URL</label>
  <div class="col-sm-10">
    <input type="text"
           class="form-control"
           id="url"
           name="url"
           placeholder="Link URL"
           [(ngModel)]="newPin.url">
  </div>
</div>

<div class="form-group">
  <label for="url"
          class="col-sm-2 control-label">Image URL</label>
  <div class="col-sm-10">
    <input type="text"
           class="form-control"
           id="url"
           name="url"
           placeholder="Image URL"
           [(ngModel)]="newPin.src">
  </div>
</div>

<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
```

```
        <button type="submit"
                class="btn btn-default"
                >Submit</button>
      </div>
    </div>
    <div *ngIf="saving">
      Saving...
    </div>
  </form>
```

3. AddPinComponent控制器

现在我们就可以定义AddPinComponent了。先从两个实例变量开始。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
export class AddPinComponent {
  saving: boolean = false;
  newPin: Pin;
```

saving会告诉用户我们正在进行保存，而newPin用于存储我们正在使用的Pin对象。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
  constructor(@Inject('PinsService') private pinsService: PinsService,
              @Inject('$state') private uiState: IStateService) {
    this.newPin = this.makeNewPin();
  }
```

如前所述，我们用Inject在constructor中注入了这些服务，并且把this.newPin的值设置成了makeNewPin，也就是下面这个函数。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
  makeNewPin(): Pin {
    return {
      title: 'Steampunk Cat',
      description: 'A cat wearing goggles',
      user_name: 'me',
      avatar_src: 'images/avatars/me.jpg',
      src: '/images/pins/cat.jpg',
      url: 'http://cats.com',
      faved: false,
      id: Math.floor(Math.random() * 10000).toString()
    };
  }
```

这看起来很像AngularJS中的定义方式，不过这种方式现在的优点在于它是带类型信息的。

当用户提交表单时，我们调用onSubmit，其定义如下所示。

code/conversion/hybrid/ts/components/AddPinComponent.ts

```
  onSubmit(): void {
    this.saving = true;
```

```
    console.log('submitted', this.newPin);
    setTimeout(() => {
      this.pinsService.addPin(this.newPin).then(() => {
        this.newPin = this.makeNewPin();
        this.saving = false;
        this.uiState.go('home');
      });
    }, 2000);
  }
```

我们再次使用超时（timeout）技术来模拟通过向服务器发起请求来保存图钉的效果。这里我们使用的是`setTimeout`。下面对比一下在AngularJS中实现同样功能的写法。

code/conversion/AngularJS/js/app.js

```
  ctrl.submitPin = function() {
    ctrl.saving = true;
    $timeout(function() {
      PinsService.addPin(ctrl.newPin).then(function() {
        ctrl.newPin = makeNewPin();
        ctrl.saving = false;
        $state.go('home');
      });
    }, 2000);
  }
```

注意，我们在AngularJS中必须使用`$timeout`服务。为什么呢？这是因为AngularJS是基于摘要循环（digest loop）的。如果你在AngularJS中直接使用`setTimeout`，那么当调用回调函数时，它会处于Angular的控制范围之外。因此改动造成的影响不会扩散出来，除非某些代码触发了摘要循环（比如使用`$scope.apply`）。

然而在Angular中，你可以直接使用`setTimeout`，因为Angular中的变更检测使用的是Zones，所以更加自动化。你再也不用担心摘要循环了，这太好了！

在`onSubmit`中，我们通过下列代码调用了`PinsService`：

```
this.pinsService.addPin(this.newPin).then(() => {
// ...
```

`PinsService`可以通过`this.pinsService`来访问，因为我们定义`constructor`时使用了特殊写法。编译器没有报错，这是因为我们已经在app.d.ts中声明过`addPin`接收一个`Pin`对象作为第一个参数。

code/conversion/hybrid/js/app.d.ts

```
  export interface PinsService {
    pins(): Promise<Pin[]>;
    addPin(pin: Pin): Promise<any>;
  }
```

我们还把`this.newPin`定义成了一个`Pin`对象。

16

addPin解析完成后，我们把this.newPin重置为this.makeNewPin()的结果，并设置this.saving = false。

要返回首页，就要使用ui-router的$state服务。我们已经通过依赖注入把它存储到了this.uiState属性中，所以可以直接调用this.uiState.go('home')来变更状态。

16.6.11　使用 AddPinComponent

我们现在就来使用AddPinComponent。

1. 降级Angular的AddPinComponent

要想使用AddPinComponent，就得先把它降级。

code/conversion/hybrid/ts/app.ts

```
angular.module('interestApp')
  .directive('pinControls',
             upgradeAdapter.downgradeNg2Component(PinControlsComponent))
  .directive('addPin',
             upgradeAdapter.downgradeNg2Component(AddPinComponent));
```

这会在AngularJS中创建一个addPin指令，它会匹配<add-pin>标签。

2. 路由到add-pin

为了使用这个新的AddPinComponent页，就要把它放进AngularJS应用中的某个地方。这很简单，只要让路由器拿到这个add状态，并把<add-pin>指令放到模板中就可以了。

code/conversion/hybrid/js/app.js

```
    .state('add', {
      template: "<add-pin></add-pin>",
      url: '/add',
      resolve: {
        'pins': function(PinsService) {
          return PinsService.pins();
        }
      }
    })
```

16.6.12　把 Angular 的服务暴露给 AngularJS

目前，我们已经降级了Angular的组件使其能用在AngularJS中，还升级了AngularJS的服务使其能用在Angular中。但是当我们的应用开始升级到Angular时，可能会需要用TypeScript/Angular写一些服务，并把它暴露给AngularJS的代码。

那么我们就在Angular中创建一个简单的“分析”（analytics）服务，用来记录事件。

我们的想法是：在应用中有一个AnalyticsService，我们将调用它的recordEvent方法。在

具体实现上，我们只会调用console.log来记录该事件，并把它存到一个数组中。这样做是为了把精力集中在最重要的事情上：描述如何把Angular的服务共享给AngularJS。

16.6.13 实现 AnalyticsService

我们先来看看AnalyticsService的实现。

code/conversion/hybrid/ts/services/AnalyticsService.ts

```
import { Injectable } from '@angular/core';

/**
 * Analytics Service records metrics about what the user is doing
 */
@Injectable()
export class AnalyticsService {
  events: string[] = [];

  public recordEvent(event: string): void {
    console.log(`Event: ${event}`);
    this.events.push(event);
  }
}

export var analyticsServiceInjectables: Array<any> = [
  { provide: AnalyticsService, useClass: AnalyticsService }
];
```

这里需要注意两点：recordEvent和Injectable。

recordEvent很简明：我们接收一个event: string参数，输出它的日志，并且把它保存到events中。在现实世界的应用中，你可能会把它发给某个外部服务，比如Google分析或Mixpanel。

要让该服务可注入，我们得做两件事：(1) 为该类添加@Injectable注解；(2) 把AnalyticsService这个令牌bind到该类。

现在，Angular将会管理该服务的单例对象，而我们可以把它注入到任何需要它的地方了。

16.6.14 把 Angular 的 AnalyticsService 降级到 AngularJS

在AngularJS中使用AnalyticsService服务之前，我们需要把它降级。

把Angular服务降级到AngularJS的过程和指令的降级过程很相似，只不过多出了一个额外的步骤：得先确保AnayticsService出现在了我们这个NgModule的providers列表中。

code/conversion/hybrid/ts/app.ts

```
@NgModule({
  declarations: [
```

```
    PinControlsComponent,
    AddPinComponent
  ],
  imports: [
    CommonModule,
    BrowserModule,
    FormsModule
  ],
  providers: [
    AnalyticsService,
  ]
})
class InterestAppModule { }
```

然后就可以使用downgradeNg2Provider了。

code/conversion/hybrid/ts/app.ts

```
angular.module('interestApp')
  .factory('AnalyticsService',
           upgradeAdapter.downgradeNg2Provider(AnalyticsService));
```

我们先调用angular.module('interestApp')来取得AngularJS的模块，然后像在AngularJS中一样调用.factory。要想降级该服务，要调用upgradeAdapter.downgradeNg2Provider(AnalyticsService)。它会把我们的AnalyticsService包装到一个函数中，而该函数会把它适配成一个AngularJS的工厂（factory）。

16.6.15　在 AngularJS 中使用 AnalyticsService

现在就可以把Angular的AnalyticsService注入到AngularJS中去了。假如我们想记录HomeController是什么时候被访问的，就可以像下面这样来记录此事件。

code/conversion/hybrid/js/app.js

```
.controller('HomeController', function(pins, AnalyticsService) {
  AnalyticsService.recordEvent('HomeControllerVisited');
  this.pins = pins;
})
```

这里注入了AnalyticsService，就像它是AngularJS中的普通服务一样，然后调用recordEvent。真棒！

我们可以在AngularJS中任何能使用依赖注入的地方使用该服务。比如，我们也可以把AnalyticsService注入到AngularJS的pin指令中。

code/conversion/hybrid/js/app.js

```
.directive('pin', function(AnalyticsService) {
  return {
    restrict: 'E',
    templateUrl: '/templates/pin.html',
```

```
    scope: {
      'pin': "=item"
    },
    link: function(scope, elem, attrs) {
      scope.toggleFav = function() {
        AnalyticsService.recordEvent('PinFaved');
        scope.pin.faved = !scope.pin.faved;
      }
    }
  }
})
```

16.7 总结

现在我们掌握了把AngularJS应用升级到AngularJS/Angular混合式应用时所需的工具。AngularJS和Angular之间也有非常好的互操作性，这是因为Angular开发组付出了很多努力来对其进行简化。

AngularJS和Angular的指令与服务之间能够互通，让应用升级变得非常容易。当然，我们不可能一夜之间就把AngularJS的应用升级到Angular，不过`UpgradeAdapter`能让我们不必把那些老代码扔掉就开始使用Angular。

16.8 参考资源

如果你想了解关于混合式Angular应用的更多知识，可以参阅下列资源。

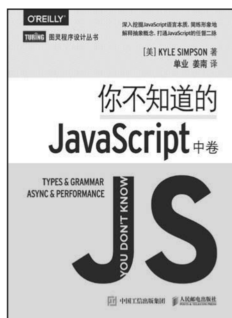
- 官方的Angular升级指南（中文版）：https://angular.cn/docs/ts/latest/guide/upgrade.html
- Angular升级模块的单元测试：https://github.com/angular/angular/blob/master/modules/angular2/test/upgrade/upgrade_spec.ts
- Angular中`DowngradeNg2ComponentAdapter`的源代码：https://github.com/angular/angular/blob/master/modules/angular2/src/upgrade/downgrade_Angular_adapter.ts

延 展 阅 读



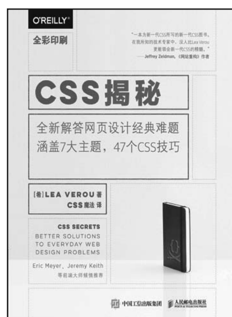
JavaScript这门语言简单易用，很容易上手，但其语言机制复杂微妙，即使是经验丰富的JavaScript开发人员，如果没有认真学习的话也无法真正理解。“你不知道的JavaScript”系列就是要让不求甚解的JavaScript开发者迎难而上，深入语言内部，弄清楚JavaScript每一个零部件的用途。

书号：978-7-115-38573-4
定价：49.00 元



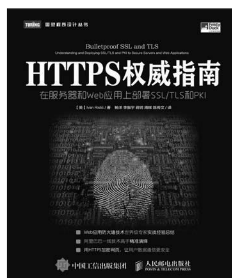
- 深入挖掘JavaScript语言本质，简练形象地解释抽象概念，打通JavaScript的任督二脉
- 2016年最受欢迎电子书 技术类TOP10

书号：978-7-115-43116-5
定价：79.00 元



- CSS一姐Lea Verou作品
- 近年来最重要的CSS技术书
- 全新解答网页设计经典难题

书号：978-7-115-41694-0
定价：99.00 元



- Web应用防火墙技术世界级专家实战经验总结
- 阿里巴巴一线技术高手精准演绎
- 用HTTPS加密网页，让用户数据通信更安全

书号：978-7-115-43272-8
定价：99.00 元



微信连接



回复“前端”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

ng-book 2 The Complete Book on Angular 2

“很高兴这本《Angular权威教程》成为Angular中文资源的一部分，希望它能广受欢迎，给中国的Angular社区提供一份令人愉悦的学习资源，也希望它帮助更多工程师开始使用下一代Angular框架来开发应用。”

——Naomi Black，Google Angular项目经理兼主管

“作为一项开源技术和前沿Web开发框架，Angular持续吸引着中国区开发人员的关注。作为雪狼及其所属Nice Angular社区的集体工作成果，这本书是开源力量的又一次证明，证明这种热情、这种志愿精神确实可以帮助业界享受到全球最新的开发技术。”

——栾跃，Google开发技术推广部大中华区主管

“作者们太棒了！如果没有这本书，真不知道我该怎么学习Angular。你们让学习并跟进Angular变得更简单了。再次感谢！”

——Jacob Cheriathundam，AccountsPRO公司CTO、高级开发工程师兼开发架构师

“我刚刚读完这本书，认为它是目前学习Angular的最佳材料。”

——Jegor Uglov，BlaBlaBlogger产品主管

“如果你和我一样是一名经验丰富的开发者，并且在积极寻找关于Angular最新信息的高效来源，那就别再找了！这本书就是目前最棒的参考资料，简洁易懂、结构合理。”

——Frederic Filiatrault，TEKsystem公司高级软件工程师

“我在书中获取了大量有价值的信息，而这是在其他网络资源中无法做到的。在我深入这些前沿工具和主题的时候，这本书给了我极大帮助。”

——Sean McGill，Anexinet公司高级顾问

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/Web开发/Angular

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-45158-3
定价：109.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks